National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

UNIVERSITY OF ALBERTA

Parallel Depth-First Search

by

Sam Farrage

A thesis

submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Edmonton, Alberta

Fall 1991

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-69952-3

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

Name of author: Sam Farrage

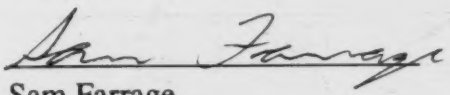
Title of thesis: Parallel Depth-First Search

Degree: Master of Science

Year this degree granted: 1991

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



Sam Farrage
1897 Barlee Road,
Kelowna, British Columbia,
Canada, V1Y 4S3

Date: SEPT. 27, 1991

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance; a thesis entitled **Parallel Depth-First Search** submitted by **Sam Farrage** in partial fulfillment of the requirements for the degree of **Master of Science**.

T. A. Marsland (Supervisor)

J. Schaeffer

P. van Beek

E. Nyland (Physics)

Date: AUGUST 6, 1991

ABSTRACT

A general-purpose algorithm for distributed depth-first search is described. The main feature of the algorithm, unlike many others, is that idle processors actively attempt to find work, instead of simply waiting for work to be assigned to them. This feature ensures that idle processors find work quickly and that excessively many tasks are not generated. The algorithm is also decentralized: there is no single processor (or small group of processors) that can become overloaded because it must manage the entire system, thus making this technique suitable for systems with many processors. Depending on the problem domain, processors have little or no responsibility for other processors' activities. Finally, it is not necessary to maintain elaborate data structures for keeping track of where work is sent.

The algorithm's efficiency depends mainly on the amount and, in some domains, also the location of work that can be transferred at any one time. The effects of changing the sizes of the work loads are investigated by testing the algorithm on a simple heuristic search problem. In the example problem domain, the sequential and parallel algorithms search the same number of nodes, thus separating the unpredictable and non-repeatable effects of speedup anomaly from those of the overheads associated with work distribution, thereby increasing the accuracy of the speedup calculations.

The results show that the type of asynchronous algorithm described here is efficient and that there is considerable freedom in choosing the method of work distribution. Modifications that deal with the specific details of more complex search algorithms, such as branch-and-bound optimization and alpha-beta game-tree search, are also discussed.

ACKNOWLEDGEMENTS

I would like to thank Dr. T.A. Marsland for supervising my thesis and for providing many useful references and suggestions. Visiting Professor D. Beal also made several helpful suggestions. I would also like to acknowledge the members of my examining committee, Dr. J. Schaeffer, Dr. P. van Beek and Dr. E. Nyland and the committee chairman, Dr. K. Smillie for the encouragement that they also gave me.

Table of Contents

Chapter 1: Introduction 1

Chapter 2: Basic Search Techniques 5

2.1: Types of Search 7

2.2: Breadth-First Search 7

2.3: Depth-First Search 8

2.4: Depth-First Iterative-Deepening 10

2.5: Heuristic Search 11

2.6: Best-First Search 12

2.7: Iterative-Deepening A* 12

2.8: Depth-First Branch-and-Bound 13

2.9: Minimax Algorithms 14

2.10: Other Algorithms 14

2.11: Applications 15

Chapter 3: Parallelism 17

3.1: Parallel Overheads 18

3.1.1: Synchronization 18

3.1.2: Data Access 18

3.1.3: Information Deficiency 19

3.1.4: Work Sharing 20

3.2: Architectures and Granularity 20

3.3: Types of Parallel Algorithms 22

3.4: Speedup 22

3.5: Amdahl Effect 23

3.6: Speedup Anomaly 24

Chapter 4: Parallelization of Depth-First Search 27

4.1: Methods 27

4.2: Previous Results 29

4.3: Description of Algorithm 33

4.3.1: Major Overheads 38

4.4: Implementation 41

4.4.1: Work Distribution 41

4.4.2: Other Work Distribution Strategies 44

4.4.3: Details of Work Distribution 45

4.4.4: Interrupts 49

4.4.5: Termination Detection 50

4.4.6: Deadlock Prevention 52

4.4.7: Effects of Increased Node Expansion Cost 52

Chapter 5: Experiments 54

5.1: Problem Domain 55

5.2: Expectations 57

5.3: Difficulties 58

5.4: Results 60

5.4.1: Effect of *MAXDEPTH* 67

5.4.2: Effect of *MINDEPTH* 69

5.4.3: Significance of *MAXDEPTH–MINDEPTH* 70

5.4.4: Discussion 71

5.4.5: Other Results 72

5.5: Different-Sized Configurations and Problems 73

5.6: Observations 79

5.7: Conclusions 81

Chapter 6: Conclusions and Future Additions 83

6.1: Larger Configurations and Connectivity 83

6.2: Better Allocation of Work 83

6.3: Reducing the Node Expansion Cost 84

6.4: Transpositions 86

6.5: Other Search Algorithms 88

6.5.1: IDA* 89

6.5.2: Branch-and-Bound 90

6.5.3: Alpha-Beta 90

6.5.4: Other Algorithms 93

6.6: Generalization 93

6.7: Fault Tolerance 94

6.8: Loaded and Heterogeneous Systems 95

6.9: Conclusions 96

References 98

List of Tables

Table 1: Average times for problem 66 61

Table 2: Average times for problem 88 62

Table 3: Problem 66 74

Table 4: Problem 60 75

Table 5: Problem 82 76

Table 6: Problem 88 77

Table 7: Summary of times 78

Table 8: Speedups 78

List of Figures

Figure 1: A search tree 7

Figure 2: Depth-first search 10

Figure 3: Speedup anomaly is possible 25

Figure 4: Searching functions (all processors) 36

Figure 5: Main program (all processors) 37

Figure 6: Choices of nodes to transfer 41

Figure 7: Search tree of sending processor (before transferring nodes) 46

Figure 8: Search tree of receiving processor (after receiving nodes) 46

Figure 9: Stacks before and after work transfer 47

Figure 10: The 15-puzzle 56

Figure 11: Time as a function of *MINDEPTH* and *MAXDEPTH* (#66) 64

Figure 12: Time as a function of *MINDEPTH* and *MAXDEPTH* (#88) 65

Figure 13: Efficiency as a function of *MAXDEPTH* and problem size 66

Figure 14: A transposition 86

CHAPTER 1

1. Introduction

Many problems in artificial intelligence and combinatorics (e.g. puzzles, games, combinatorial optimization) can be represented by a graph whose nodes represent partially solved problem states and whose arcs represent transitions between states (elementary operations). Arcs may be weighted, thus associating a cost with the corresponding transition. A problem can be solved by searching the corresponding graph, which is known as the *search space*.

One node in the search space, the *start node*, represents the initial state of the problem. Depending on the problem domain, the initial state could represent a given initial configuration or the absence of knowledge. Solutions to the problem are represented by *goal nodes*. The object of a search may be to find any goal node, to find all goal nodes, or to find a goal node that is *optimal* according to some domain-dependent criteria (e.g. the shortest path from the start node or according to an evaluation function).

The search space can be treated as a tree by ignoring the fact that identical nodes can be generated several times, via different paths. For some graphs, ignoring this repetition (treating identical nodes like separate ones) results in much extra search; for others, the extra search requires less time than detecting the duplicated nodes. The latter is assumed to be true in this thesis, since detecting repeated nodes is a major issue in itself, and the search space is treated as if it were a tree. It is also assumed that all arcs have unit weight.

Search spaces are usually so large that it is impractical to try to search all nodes. However, it is usually not necessary to search the entire space: using information

about the problem, it is often possible to determine that a solution (of whatever type is desired) is more likely to be found in one subset of nodes than in another or even that a subset of nodes cannot contain a solution. Using such techniques, it is often possible to solve a problem in a reasonable amount of time. However, further speed-ups are often desirable: for example, for programs that have to meet deadlines or interact with a user.

Further reductions in search time can be achieved by using multiple processors simultaneously (*parallelism*), in addition to the search reduction techniques just described. Unfortunately, developing parallel algorithms is not a trivial matter because there are numerous problems and overheads that are not encountered in sequential algorithms. For example, processors need to communicate to distribute data; communication causes delays and requires extra programming. Processors often finish their work and remain idle until they can obtain more work. Some processors may lack information that would be available to a sequential program. This lack of information can cause processors to do extra (often redundant) work that the sequential algorithm avoids, often greatly reducing the advantages of the search reduction techniques previously described. In addition, debugging is more difficult because many errors do not repeat between runs. The occurrence of certain errors depends on precise timing, which may be affected by unrelated events, such as the activities of other users.

The goal of this thesis is to investigate an effective, general technique for parallelizing *depth-first search* (see Section 2.3) on systems with many processors. To be effective in large systems, it is necessary to eliminate situations in which many processors have to wait for service from one processor (for example, when only one processor controls the distribution of work). It is also helpful if processors can proceed independently without having to keep track of computations done by other processors

because the complexity of maintaining this information usually increases quickly (more than linearly) as the number of processors increases.

The thesis opens by describing depth-first search, as well as alternative and specialized search techniques, and follows with a discussion of issues related to parallelism. The main contribution of the thesis is the description, in Chapter 4, of a program named PDFS (which stands for *Parallel Depth-First Search*). In PDFS, idle processors actively attempt to find work, instead of waiting for work to be assigned to them. Idle processors seek work from all other processors, rather than from a single source, which may become a bottleneck. Busy processors normally search the tree as a sequential processor would, without worrying about the activities of any other processors. They interrupt their searching only when messages arrive from other processors; after quickly taking action, they continue searching.

Chapter 5 presents experiments in which PDFS was tested on a simple search problem. A simple problem domain was chosen because the results are more general than for algorithms that must deal with the specific complexities of more difficult problems. It is easier to use a simple algorithm as a starting point for more complex searches, such as IDA* or branch-and-bound, than to modify a complex algorithm to solve a totally different problem. Attempting to solve one particular problem is not the goal of this thesis: trying to solving one problem is less helpful in developing general techniques for parallel search than is running controlled experiments with a simple, general algorithm. Speedup anomaly (see Section 3.6) was also avoided because its effects are unpredictable.

PDFS's efficiency depends on several factors; one of the most important is the size of work loads that are distributed when processors become idle. The location of these work loads in the tree can also be an important factor in some problem domains. The size of the work loads is controlled by two parameters; the effect of varying these

parameters was investigated. The experiments show that, for a wide range of parameter values, the performance was good, and little time is wasted while processors look for work. Using a set of parameter values in this range, PDFS was then tested on different-sized problems and processor configurations.

Because of the simplicity of the search domain, the speedups are expected to be better than for more complex domains. However, simple domains give better insights into how to improve performance than do complex domains, because simpler algorithms are less burdened by problem-specific details, which can reduce performance.

The thesis closes with a discussion of future extensions to PDFS, including how it can be improved to take advantage of much larger numbers of processors, how it can be modified to perform more specific, complex tasks and to take advantage of the properties of specific algorithms, how to detect repeated nodes to avoid redundant searching, and how to recover if one or more processors fail.

CHAPTER 2

2. Basic Search Techniques

There are many methods for searching graphs and trees. Which method is best depends on the problem domain and the purpose of the search. For example, the search might try to find one solution, all solutions, or an *optimal* solution. There are many possible criteria that can be used to choose a specific node from a set of goal nodes. Two common examples are the shortest path from the start node to the goal node (*optimization* or *least cost*), and the goal node that maximizes or minimizes some evaluation function (e.g. alpha-beta [9]). In the example used here, a goal node (solution) is optimal if there is no other goal node that is closer to the start node. (There may be more than one optimal solution, since two goal states may have equal cost, where cost is defined as the sum of the weights of the arcs along the path from the start node to the goal node.) This example is a special (but frequently desired) case of least-cost, in which all transitions have unit cost (i.e. all arcs have unit weight).

In this thesis, it is assumed that the search graph is a tree and that all arcs have unit weight. The following definitions are needed:

- A *successor* of a node is a node that can be reached by performing one elementary operation on that node. For example, in Figure 1, B, C and D are successors of A.
- Successors of the same node are *siblings* of one another.
- A node that is searched before one of its siblings is said to be *older* than that sibling.

- A node is the *parent* of its immediate successors.
- A node that has no successors is a *leaf* or *terminal* node. For example, in Figure 1, E through M are leaf nodes.
- A *descendant* of a node is defined recursively as a successor or a descendant of a successor.
- An *ancestor* of a node is the parent or an ancestor of the parent.
- A tree's *root* node is the ancestor common to all nodes in the tree. For example, in Figure 1, node A is the root of the tree.
- The *depth* of a node is defined as follows: the root node's depth is zero; if a node has depth d, its successors have depth $d+1$. A node at depth d has d ancestors.
- A *subtree* of another tree consists of a node in the tree (the root of the subtree) plus all of its descendants.
- The current *search path* is the path from the root node to the node currently being searched.
- The *average branching factor* of a tree is the average number of successors of the nodes.
- A *uniform* tree with branching factor b is a tree in which all terminal nodes occur at the same depth and all non-terminal nodes have b successors. There are b^d nodes at depth d in a uniform tree.
- *Expanding* a node means generating its successors, as well as performing the usual operations involved in searching (e.g. evaluation). Since search spaces are usually too large to store in memory, nodes are generated only when they are needed.

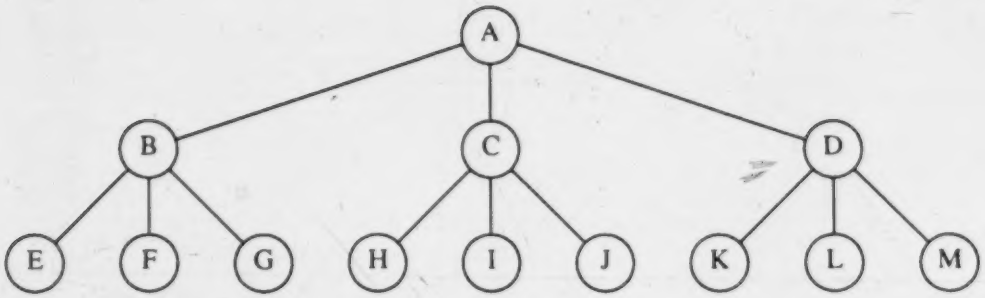


Figure 1: A search tree

2.1. Types of Search

There are two main types of search: *best-first* and *depth-first*. The main difference between them is the method used to determine the next node to be expanded. In best-first search, there is an ordered list, to which nodes are added as they are generated. The next node to be expanded is chosen from the list according to certain criteria (e.g. heuristic functions). In depth-first search, the next node to be expanded is the first successor (of the current node) that has not been expanded already. In the following discussions, it is assumed that the object of the search is to find the goal node nearest to the root node.

2.2. Breadth-First Search

One way to find the shallowest goal node is to start by searching the root node, then to search all nodes at depth 1, then all nodes at depth 2, etc. For each d, all nodes at depth d are searched before any at depth $d+1$. This method, known as *breadth-first search* (a type of *best-first* search, which is discussed later), guarantees that the first goal node found will be optimal (by the definition used earlier).

Breadth-first search has several drawbacks. The main problem is that all nodes at depth d must be stored in memory, or otherwise regenerated, before any nodes at depth $d+1$ can be searched. Since the number of nodes at depth d usually increases exponentially with respect to d, breadth-first search usually exhausts available memory before a solution can be found. Also, storing and retrieving or regenerating nodes increases the cost of each node expansion.

Breadth-first search would search the nodes in the tree in Figure 1 in the following order:

A, B, C, D, E, F, G, H, I, J, K, L, M

Information about nodes B, C and D would usually have to be stored before nodes E to M could be generated, unless B, C and D could be regenerated somehow (e.g. by iterative-deepening, which is discussed in Section 2.4).

2.3. Depth-First Search

In *depth-first search*, all of a node's descendants are searched before any of its younger siblings (see Figure 2). Nodes in the tree in Figure 1 would be searched in the following order:

A, B, E, F, G, C, H, I, J, D, K, L, M

This ordering, also known as an *in-order traversal* of the tree, causes several problems that do not occur in breadth-first search. The first goal node found is not necessarily optimal: a younger, and therefore unsearched, sibling of an ancestor of the current node may be a goal node. For example, if D and G are goal nodes, depth-first search finds G first, even though D is better. There is no guarantee that any specific subtree even contains a goal node. Since subtrees may be large, it is not feasible to attempt to search all subtrees completely as they are encountered.

Unless the search space is small, some criteria are needed to limit the search by deciding when to stop searching and try another path (*backtrack*). One such limiting method is to stop searching nodes deeper than a predetermined or computed depth, d, as was done in the above example. Unfortunately, d_S, the depth of the optimal solution, is rarely known in advance. If $d_S > d$, no solution will be found because all goal nodes are deeper than the limiting depth; the search effort would then be wasted. If $d_S < d$, a sub-optimal solution, whose depth is greater than d_S, may be found first. It would then be necessary to continue looking for an optimal solution, resulting in a search of all nodes with depths less than or equal to d_S, as well as many nodes with depths greater than d_S. Having to search these extra nodes is a problem because there are about b (the branching factor) times as many nodes at depth $d+1$ than at depth d. When the program decides that it can ignore a subtree, a *cutoff* is said to occur.

The main advantage of depth-first over breadth-first search is that the memory requirements are linear in the depth of the search: only the direct ancestors of the node being searched need to be stored (in a stack); it is not necessary to store entire levels of the tree. Also, since it is not necessary to manage a large data structure, the cost of searching an individual node is lower for depth-first search than for breadth-first. This saving is often so great that depth-first search is faster than breadth-first, even though it usually searches more nodes.

Figure 2 shows pseudocode for a depth-first search algorithm that attempts to find a goal node. For efficiency, successor states are often generated incrementally by modifying a global data structure that represents the current state, instead of explicitly generating new states. Depending on the problem domain, extra operations are often required and other arguments (e.g. the last move or the cost of the current path) often need to be passed to the function.

```
search(n)        /* n = current state */
    if (cutoff_condition(n))    /* backtrack */
        return(NOT_SOLVED)
    if (goal(n))                /* problem solved */
        return(SOLVED)
    generate successors s1, ..., sk of n
    for i=1 to k
        if search(si)==SOLVED
            return(SOLVED)
    return(NOT_SOLVED)
end
```

Figure 2: Depth-first search

2.4. Depth-First Iterative-Deepening

Korf [10] uses *iterative-deepening*, a technique for improving move ordering in game-tree search [24, page 101], to find an optimal solution without having to search deeper nodes or store entire levels of the tree in memory. The idea is to do repeated depth-first searches: first search the root, then all nodes with *depth* ≤1, then all nodes with *depth* ≤2, and so on, until a solution is found. The tree shown in Figure 1 would be searched as follows (three iterations, separated by semicolons):

A; A, B, C, D; A, B, E, F, G, C, H, I, J, D, K, L, M;

As in breadth-first search, all nodes at depth d are searched before any nodes at depth $d+1$, so the first solution found will be optimal. However, since the search is depth-first, it is not necessary to store entire levels of the tree in memory; a stack is sufficient. Thus, iterative-deepening finds optimal solutions, preserves the low memory requirements and node expansion cost of depth-first search, and avoids searching nodes deeper than the solution, thus eliminating the possibility of performing an exponential amount of unnecessary search.

The main problem with iterative-deepening is that shallower nodes are searched many times. This extra searching is not a serious problem if the tree has a high

branching factor. If the branching factor is too low, however, the nodes searched in previous iterations make up a large proportion of the total nodes searched (i.e. most of the time is spent redoing work).

In some applications (e.g. two-player games), the order in which successors are explored affects the total amount of search required. Information obtained in previous searches can be used to reorder successors, thus reducing the amount of search done in the next iteration [24, page 101].

2.5. Heuristic Search

It is often possible to determine in advance that it is not necessary to search some nodes, or that some paths are more likely to lead to solutions than others. One way to make these determinations is to use a problem-dependent *heuristic* function [19] that estimates the distance from any node to a goal node. The following symbols are used throughout this thesis:

$g(n)$ is the cost of the path from the start node to n.

$h^*(n)$ is the cost of the shortest path from n to a goal node.

$h(n)$ is a heuristic function that estimates $h^*(n)$.

$f(n) = g(n) + h(n)$ is the estimated cost of a path through n from the start node to a goal node.

A heuristic search algorithm is *admissible* if it always finds an optimal (least-cost) path to a solution (before finding any sub-optimal paths). The term *admissible* is also used to refer to heuristics for which this condition holds.

2.6. Best-First Search

It is often desirable to choose the most promising node (as determined by a heuristic function) as the next one to be searched. In best-first search, all nodes generated but not yet searched are stored in an ordered list (usually called the *OPEN* list). Best-first search repeatedly expands (an operation that includes removal from *OPEN*) the best node, as predicted by the heuristic function, until a solution is found or all nodes are searched. Best-first search suffers from the same problems as breadth-first search: excessive memory requirements and the high overheads of managing the list. (Breadth-first search is an example of an admissible best-first search, since $h(n) = 0 \leq h^*(n)$ for all nodes n: the "best" node is one at the shallowest depth that has not been searched completely.)

An example of best-first search is A* [17, Chapter 3, pages 57-59] [19, Chapter 2, page 64]. A* also uses a *CLOSED* list, which prevents identical nodes from being searched repeatedly; it includes all the information that would be present in a complete transposition table (see Chapter 6). The *CLOSED* list also keeps track of the parents of nodes, thus allowing the complete path from the root node to the goal node to be reconstructed. Nilsson [17, Chapter 3] proves that if $h(n) \leq h^*(n)$ for all n, A* is admissible.

2.7. Iterative-Deepening A*

Depth-first iterative-deepening can also be applied to heuristic search. Instead of cutting off the search when the depth, g, exceeds the threshold, the search is cut off when $g + h > threshold$. Korf [10] used the iterative-deepening A* algorithm (IDA*) to solve the memory problems of A* by using repeated depth-first searches, with increasing thresholds to search for solutions. The initial threshold is $h(root)$. After each unsuccessful iteration, the smallest value of $f(n)$ (evaluation of a node) that

caused a cutoff becomes the next threshold. The first successful iteration will have a threshold equal to the cost of the optimal solution, $g(s)$ (which equals $h^*(\textit{start node})$). If h is admissible, this threshold will be encountered eventually because $h(s) = h^*(s) = 0$, which implies that $f(s) = g(s)$.

Because IDA* is a depth-first search, its memory requirements are only linear with respect to the depth of the optimal solution and there is no large data structure to manage. However, as described in Section 2.4, one deficiency of IDA* is that non-leaf nodes are searched repeatedly. All searching done with thresholds less than the solution cost is wasted. For problems with large branching factors, the amount of repetition is small compared to the total amount work done because as the threshold increases, the number of nodes evaluating to that threshold grows rapidly. However, even for the 15-puzzle (see Section 5.1), which has an average branching factor of only 2, the extra cost (in each node expansion) of managing the *OPEN* and *CLOSED* lists in A* exceeds the cost of the extra node expansions in IDA*.

However, IDA* is ineffective if the threshold changes frequently because almost all the work done in an iteration is a repetition of the previous iteration. Few new nodes are searched, since most of the cutoffs in the new iteration occur at exactly the same places as before. The total amount of repetition is huge, since the number of iterations required is equal to the number of times the threshold changes. The traveling salesman problem [13] is an example for which IDA* is ineffective. Frequent occurrence of duplicated states (see Section 6.4) also reduces the effectiveness of IDA*.

2.8. Depth-First Branch-and-Bound

Another way to find an optimal path to a solution is by trying to search the entire tree, but cutting off the search when the cost exceeds that of the best path found so

far. This technique can be used in problems (e.g. traveling salesman problem) in which all solutions occur at the same depth, but an optimal one must be chosen from among them.

2.9. Minimax Algorithms

Algorithms that compute the *minimax* value of a tree can be used in two-player games, in which one player tries to maximize the score while his opponent tries to minimize it. The minimax value of a leaf node is determined by an evaluation function. The minimax value of an even-depth, non-leaf node is the maximum of the minimax values of its successors; for an odd-depth node, it is the minimum. Nodes beyond a certain depth are treated as leaf nodes, even if they actually do have successors, because the size of most search trees makes it impractical to try to search the entire tree.

Alpha-beta [9] is a depth-first branch-and-bound algorithm that uses two bounds, α and β to eliminate much of the search required by an exhaustive minimax search algorithm. SSS* [25] is a best-first algorithm that finds the minimax value of a tree. When modified as Campbell and Marsland specify [3, pages 354-355], SSS* always examines only a subset of the leaf nodes examined by alpha-beta. SSS* suffers from the same excessive memory requirements and increased node expansion cost as other best-first search algorithms.

2.10. Other Algorithms

Since best-first search requires too much memory and depth-first search often expands too many nodes, combining the two methods may yield a better algorithm. Best-first search can be used until memory is exhausted, then depth-first search can be applied to all subtrees rooted at nodes in the *OPEN* list. The latter part of the search

is easy to parallelize (see Chapter 3) because it consists of many independent tasks. Campbell and Marsland [3, pages 357-358] suggest several other forms of this *staged search*, such as having several stages of best-first search.

Also, it is possible that a non-admissible heuristic can solve a problem more quickly, but it does not guarantee that the first solution found will be optimal. Pearl [19, Chapter 7] discusses the effects of non-admissible heuristics. Other techniques, such as the use of macro operators [11], can also find non-optimal solutions quickly.

2.11. Applications

Although this thesis focuses on depth-first search from the viewpoint of applications in computer science, it should be realized that these techniques have applications in other areas as well.

One application of depth-first search to areas outside computer science is in applications that proceed by trial and error because direct algorithms are not known or because complete information is not available initially. In such applications, one frequently encounters dead ends and must backtrack and try alternatives. For example, when integrating a function symbolically (as opposed to numerical approximation), one might find that the current sequence of transformations leads to a function that is harder to integrate than the original one. When solving various types of mathematical problems with many variables, one might want to assign temporary values to some variables, then change them later, after determining that these values are incorrect or inferior. In chemistry, when analyzing a substance, one might find that the initial hypothesis about the likely content was incorrect, thus requiring extra or different tests. When attempting to make a schedule (e.g. a class schedule for a university or an autonomous robot trying to schedule time-limited tasks), it is often impossible to add required items, thus forcing backtracking. In all these examples,

different alternatives correspond to sibling nodes. Deciding to abandon the current sequence of steps corresponds to backtracking in depth-first search.

The techniques described in this thesis can also be applied to problems that can repeatedly be divided into independent subproblems, which can themselves be divided further. However, two problems can be solved simultaneously only if each problem can be solved without knowing the solution to the other problem. Many recursive algorithms (in computer science and in other fields) that contain multiple recursive calls fall into this category, even though such algorithms are generally not considered to be depth-first search.

CHAPTER 3

3. Parallelism

One way to solve problems in less time is by using *parallelism*: doing some operations simultaneously instead of sequentially. However, many difficulties are encountered when developing parallel programs:

- Work must be allocated to processors.
- Processors may have to wait for results from other processors.
- Processors may become idle.
- Deadlock is possible.
- Data may be inaccessible to some processors, thus causing them to do unnecessary work.
- Extra code is required for communication.
- Extra data structures must often be maintained.
- Debugging is more difficult because of non-repeatable errors.

Consequently, parallel programs are more difficult to develop than sequential programs.

The decision about which operations should proceed simultaneously is not trivial: it depends on the overheads (see Section 3.1) involved in the parallelization, which, in turn, depend on the problem domain and the architecture of the machine being used. A unit of work done by one processor in a parallel environment is called a *task*. Tasks may split into *subtasks*, which may themselves split into smaller subtasks; these subtasks are called *siblings* of each other.

3.1. Parallel Overheads

Parallel programs often have to do extra computations that a sequential algorithm would not need to do. In addition, time may be wasted because processors have no work to do, or compete with other processors for resources. These extra computations and delays, known as *overheads*, reduce the speedups (see Section 3.4), often substantially. To design an efficient program, it is necessary to reduce these overheads; unfortunately, reducing one overhead often increases another. In addition, extra instructions (*parallelization overhead*) are usually necessary to handle operations that are not done by the sequential program.

3.1.1. Synchronization

It is sometimes necessary for one processor to wait for another to perform some action or return some result. This waiting is called *synchronization*. For example, in many parallel programs, one task splits into multiple parallel subtasks, and must wait until these subtasks finish. In addition, some of these tasks may finish before others, and therefore must wait also. Synchronization overhead increases as the number of tasks increases; it also increases if the tasks differ substantially in size.

Synchronization overhead can be reduced by trying to assign equal-sized loads to processors, though it is difficult to determine the size of a task in advance. If there are more tasks than processors, a processor that finishes one task should immediately start on another one.

3.1.2. Data Access

In a parallel environment, processors do not always have immediate access to data, as they do in a sequential environment. When two processors try to access the

same memory location in a shared-memory system, the hardware often delays one of them. In addition, *mutual exclusion* is necessary for some operations, such as modifying shared data structures. Mutual exclusion means allowing only one processor to access an object (file, shared-memory location, I/O device, etc.) at a time, thereby forcing others to wait.

In a distributed system, required information may be located in another processor's memory. This data would then have to be transferred over a network, which results in two types of overhead: the delay until the data arrives and the interruption of the other processor, which must temporarily stop its current task so that it can send the data. Mutual exclusion requirements might also force a processor to delay sending data that it is currently using. Contention for the network also increases access overhead.

The overheads are much bigger in distributed systems than in shared-memory systems. However, because there is less contention for shared resources, distributed systems potentially allow more processors to operate simultaneously.

3.1.3. Information Deficiency

In distributed systems, information in one processor's local memory is not available to other processors. Another processor might be able to avoid doing certain computations if it is aware of information that another processor has. These extra computations would not be done by a sequential program. Information deficiency can be reduced by sharing new data as it is obtained, but at the expense of increased communication overhead.

3.1.4. Work Sharing

A parallel algorithm must distribute work among the processors. This distribution can be *static*, in which all work is distributed at the beginning, or *dynamic*, in which work is transferred whenever needed. When the parallel tasks finish, their results sometimes have to be combined. Lack of work is called *starvation*. Processors often need to maintain extra data structures to keep track of work that may be parallelized.

3.2. Architectures and Granularity

The size of a task is called the *granularity*. The appropriate choice of granularity depends on the architecture of the machine; a poor choice results in an inefficient parallel program.

Flynn [8] classified computer architectures according to the number of sequences (''streams'') of instructions and data that can be processed simultaneously. Most computers execute a single instruction at a time. This instruction is performed on a single set of data, hence the name SISD (Single Instruction Single Data).

Some machines allow a single instruction to be performed on many sets of data simultaneously. This type of machine is said to have a SIMD (Single Instruction Multiple Data) architecture. These machines are often called *vector* or *array* processors because they treat the data as an array, simultaneously performing the same operation on each element. Since the unit of parallelism (one instruction) is small, SIMD style computers are *fine-grained*. Such machines are well-suited for computations involving matrices.

It is possible to parallelize a wider variety of operations if different instructions can operate simultaneously on different sets of data. These MIMD (Multiple Instruc-

tion Multiple Data) style machines usually consist of multiple processors, each of which receives its own instructions and data, independently of the other processors. There are different types of MIMD architectures. Some systems have shared memory, in which all processors have the same address space and access the same physical memory. In other systems (*distributed* systems), each processor has its own memory (*local* memory), which is not directly accessible to other processors; processors must communicate by sending messages over a network or bus. Some systems have both shared and local memory.

Communication is easy in shared-memory systems, since all processors share the same address space. However, contention for this memory is a problem in large systems because hardware limitations prevent unlimited simultaneous access. Mutual exclusion also causes delays when several processors try to modify the same memory location simultaneously.

There is no contention for distributed memory, but sharing data becomes more difficult because explicit communication is required. Since network communication is more costly than shared-memory access, it is not feasible to try to share all data. The main advantage of distributed systems over shared-memory systems is that there are fewer hardware constraints, thus making larger configurations possible.

MIMD machines are more *coarse-grained* than SIMD machines: since work distribution is more costly, it must be reduced by assigning processors larger units of work, less frequently. Because of their higher communication costs, distributed systems are more coarse grained than shared memory systems, and hence must be given even larger units of work to reduce the overheads.

Sometimes part of the next instruction can be executed in parallel with the latter part of the current machine instruction. This form of parallelism, known as *pipelin-*

ing, may be present in any of the architectures just described.

3.3. Types of Parallel Algorithms

Parallel algorithms can be classified as being *synchronous* or *asynchronous*. In a synchronous algorithm, parallel tasks start and finish at specific points; computations cannot proceed until all processors have reached this point. There may be many such synchronization points in the program. One reason for synchronization is to combine results and use them to determine an appropriate distribution of work for the next parallel phase. Synchronization overhead can be large at these points because the loads assigned to sibling tasks are often unbalanced: some tasks may finish long before others, thus causing some processors to remain idle for long periods.

In an asynchronous algorithm, all processors proceed independently. Although processors may communicate with each other and often have to wait for replies to messages, they do not have to wait for other processors to finish their tasks.

Parallel algorithms may have various combinations of synchronous and asynchronous components. For example, synchronization may be needed only if certain results are obtained.

3.4. Speedup

It is usually necessary for a parallel program to perform certain extra operations related to parallelism. These extra operations slow down the parallel program.

Let T_1 be the time required by the *best* sequential algorithm.

Let T_N be time required by the parallel algorithm when using N processors.

The *speedup* of an N processor parallel algorithm is defined as $S_N = \frac{T_1}{T_N}$.

If $S_N < N$, the speedup is called *sub-linear*.

If $S_N = N$, the speedup is called *linear*.

If $S_N > N$, the speedup is called *super-linear*.

It is important to note that the *best* sequential algorithm should be used when calculating speedups, even if the parallel algorithm was derived from an inferior sequential one. Linear speedups are desirable, but are usually difficult to achieve. Super-linear speedup occurs occasionally, for some problem instances, in some algorithms. Consistent super-linear speedup implies that a better sequential algorithm could be obtained by simulating parallelism using only one processor [5], or that some operations are implicit in the parallel hardware [18].

The *efficiency* of an N-processor parallel algorithm is defined as:

$$efficiency = \frac{T_1}{NT_N}.$$

Efficiency relates the sequential time, T_1, to the *total* amount of time used by *all* parallel processors (N processors $\times T_N$). Efficiency measures the amount of time, relative to the total time, that was spent doing work equivalent to that done by the sequential algorithm. In the absence of speedup anomaly (see Section 3.6), 1–*efficiency* is the proportion of time spent on overheads.

3.5. Amdahl Effect

A certain amount of work must be done sequentially. In some circumstances, this requirement can impose an upper bound on the speedup. This limitation is known as the *Amdahl effect*. Let S be the time required to do the work that *must* be done sequentially. Let P be the time required for one processor to do all the work that can be parallelized. The time required for N processors is at least $S+P/N$. Since

an upper bound for speedups is being computed, it is assumed that the sequential algorithm requires time $S+P$, though it normally requires less. The maximum speedup is then $\frac{S+P}{S+P/N}$. As N approaches ∞, the maximum speedup approaches $\frac{S+P}{S}$.

If a fixed proportion of the work must be done sequentially, the speedup is severely limited. Let T be the total time required; let εT be the amount that must be done sequentially ($S = \varepsilon T$, $P = (1-\varepsilon)T$). As N increases, P/N approaches 0, and the maximum speedup is $\frac{S+P}{S} = \frac{T}{\varepsilon T} = \frac{1}{\varepsilon}$. For example, if 10% of the work must be done sequentially ($\varepsilon = 0.1$), the maximum speedup is 10, regardless of how many processors are used.

However, if S is a function of the problem size, n, and $S(n)/T(n)$, the proportion of work that must be performed sequentially, approaches zero as n increases, the speedup becomes:

$$\frac{S(n)+P(n)}{S(n)+P(n)/N} = \frac{T(n)}{S(n)+P(n)/N} = \frac{1}{S(n)/T(n)+P(n)/(NT(n))},$$

which, for large n, is approximately equal to:

$$\frac{1}{P(n)/(NT(n))} = \frac{1}{(T(n)-S(n))/(NT(n))} = \frac{N}{1-S(n)/T(n)},$$

which approaches N as n approaches ∞. In this case, the algorithm is said to be *effective*. For example, if the amount of work that must be done sequentially is constant, regardless of the problem size, the algorithm is effective; the Amdahl effect does not place this absolute limit on the speedup.

3.6. Speedup Anomaly

In most search domains, the sequential and parallel processors do not search

exactly the same set of nodes, or even the same number of nodes, because parallelism causes nodes to be searched in a different order. For example, one processor in a parallel algorithm may happen to find a goal node early in the search, while the sequential algorithm does not find one until most of the tree has been searched. When the numbers of nodes searched by the sequential and parallel algorithms differ only because one algorithm happens to find a goal node early in the search, *speedup anomaly* is said to occur. Speedup anomaly is most often associated with super-linear speedup.

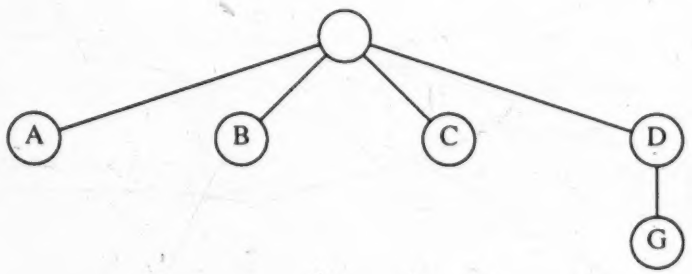


Figure 3: Speedup anomaly is possible

For example, suppose there are large subtrees rooted at nodes A, B and C in Figure 3, and that G is the only goal node. Suppose there are four processors, which are assigned nodes A, B, C and D (and their descendants) respectively. The processor that received node D would find the goal (G) immediately, whereas the sequential program (assuming left-to-right search) would have searched the three large subtrees (rooted at A, B and C) first. The other three processors could then be aborted, even if they have not finished searching their subtrees. In this example, the speedup could easily be super-linear (greater than 4, the number of processors) because the total number of nodes searched by all processors in the parallel algorithm is much less than the number of nodes searched by the sequential algorithm.

When the possibility of speedup anomaly exists, it is also possible that the sequential algorithm finds a solution early in the search, rather than late, thus resulting in lower speedups. Suppose that the only goal node is a descendant of A, and that work was assigned as in the previous case. One processor would find the solution after searching the same set of nodes as the sequential algorithm; the work done by the other processors would be wasted. The speedup would be poor, regardless of how low the overheads may be, because the parallel algorithm searches about four (the number of processors) times as many nodes as the sequential algorithm. In this example, the parallel program would take even longer than the sequential program because the parallel processors are slowed by overheads.

Speedup anomaly cannot usually be predicted in advance; otherwise both the parallel *and the sequential* algorithms could be improved by giving a higher priority to subtrees that are likely to contain goal nodes. In asynchronous programs, speedup anomaly is often non-deterministic: it changes from one execution to another, even when the same input data is used.

Examples of speedup anomaly occur when one processor in a parallel IDA* algorithm finds a solution early (in the final iteration; both algorithms search all nodes in non-final iterations) and when a parallel branch-and-bound algorithm finds (and broadcasts) a good bound that the sequential algorithm would not have found until much later in the search.

CHAPTER 4

4. Parallelization of Depth-First Search

To parallelize depth-first search, it is necessary to divide the search space into pieces (subtrees) that can be searched by different processors. This splitting may be *static*, in which the tree is split only at the beginning. Alternatively, the splitting may be *dynamic*, in which a processor's work load may be split at various times throughout the search.

The size and location of each processor's work load is an important factor in determining the efficiency of the parallelization. The major overheads are:

- work distribution: communication or shared-memory access.
- synchronization: waiting for other processors to send work.
- information deficiency, which results in extra, unnecessary searching.
- maintenance of extra data structures to support parallelism.

The problem is that reducing one overhead usually increases others. The choice of methods to reduce the total overheads depends greatly on the problem domain and the architecture of the machine being used.

4.1. Methods

In static partitioning, the tree is split at the root and the pieces allocated to all processors. This approach usually results in poor speedups because it is difficult to predict in advance how many nodes each processor will search. The branching factors and numbers of cutoffs may vary greatly among different parts of the tree. Processors that receive smaller subtrees soon finish searching, then remain idle, often for

a long time, until the others finish.

Dynamic distribution methods can be used to assign more work to processors when they become idle. In addition, it is often possible to make better choices about exactly which nodes to transfer because, as the search progresses, more is known about the structure of the tree. This information can help improve load balancing (trying to give each processor the same amount of work) by distributing work loads that are more likely to be close in size. It can also reduce excess search caused by information deficiency by not sending out (until later) nodes that are unlikely to be searched by the sequential algorithm.

One method of dynamic distribution would be to have one processor (called the *master* processor) operate as in a sequential program, except that it maintains a list of subtrees that can be searched in parallel by other processors. In a *synchronous* algorithm, the master would split into several *slave* processors after enough subtrees have been generated. Each slave, one of which would usually run on the same processor as the master, would search one subtree. When all slaves finish, the master would combine the slaves' results and either output results or continue searching to prepare for another split. In an *asynchronous* algorithm, subtrees can be sent to, and results received from, slave processors at any time, while the master continues searching. In both synchronous and asynchronous (and even some static) methods, there may be multiple levels of master-slave relationships: subtasks may split into smaller subtasks. In coarse-grained systems, asynchronous methods are usually more efficient than synchronous ones because idle processors can receive work at any time; they do not need to wait for sibling tasks to finish.

In asynchronous schemes, the decision of when to send work may be made by busy processors or by idle (requesting) ones. In the former case, a busy processor decides that certain nodes should be searched simultaneously and simply gives them

to other processors. In the latter case, a processor sends work only after becoming aware that another processor is idle (e.g. when an idle processor returns results or explicitly requests work). The latter scheme should be more efficient because busy processors immediately become aware that other processors are idle: they can send work immediately without having to guess or check at regular intervals if parallelism should be increased. It also avoids the unnecessary overheads that occur when processors are assigned several tasks at once: these tasks would have been done more efficiently if they had not been reassigned. This extra overhead should be avoided unless a processor would have remained idle otherwise.

Master-slave relationships can be made even more flexible: an idle processor can request work from *any* other processor, thus becoming its slave. The master-slave relationship ends when the slave finishes the work and returns the results to its master. In some domains, slaves do not need to return results to their masters, in which case the master-slave relationship ceases after work is transferred. This freedom increases the likelihood that an idle processor will be able to find work. In a centralized scheme, if the master becomes idle, idle slaves cannot obtain more work, whereas in a decentralized scheme, if one processor is idle, work can be requested from another. The lack of centralization makes these algorithms appropriate for extremely large systems because work distribution is not confined to a single processor, which may become overloaded.

4.2. Previous Results

This thesis focuses on asynchronous algorithms with decentralized work distribution, in which idle processors actively try to find work from all other processors. Several people have developed such algorithms for a variety of search domains. Rao, Kumar and Ramesh [21] implemented a parallel IDA* program that used such a work

distribution scheme. They used a Sequent Balance 21000, a MIMD machine with memory that is shared among all processors, to test their algorithm on the 15-puzzle (see Section 5.1). In their algorithm, one processor initially receives the entire search tree; all others are idle.

Kumar and Rao [12] implemented the same parallel IDA* algorithm on a BBN Butterfly, an Intel iPSC hypercube, and 1-ring and 2-ring configurations of processors. In the BBN Butterfly, each processor has its own memory, but can access other processors' memories through a fast switch, thus giving the appearance of shared memory. In the Intel iPSC hypercube, each processor has its own memory, which is inaccessible to other processors. Each processor in a 2^N-processor system can communicate directly with N other processors. In the ring configurations, each processor has only two neighbors: in a 1-ring, each processor can obtain work only from one of its neighbors; in a 2-ring, each processor can obtain work from either of its neighbors. The ring configurations were simulated by using processors in the iPSC, but ignoring the extra available communication paths.

Their results, also for the 15-puzzle, showed that performance decreases as communication costs increase and connectivity decreases. Their speedups for the Balance and Butterfly were almost linear (106 for 115 processors) even when more than 100 processors were used. Their speedups for the iPSC were still good (115 for 128 processors), but lower than for the Sequent and Butterfly because the iPSC, being distributed, has higher communication costs (sending and waiting for messages). The iPSC's reduced connectivity (a processor cannot communicate with every other processor) also affects performance. Kumar and Rao's speedups for the ring configurations were poor because the restricted communication paths did not allow work to be distributed efficiently: distant processors may have plenty of work, while neighboring processors may not. As expected, the 1-ring's performance was worse

than the 2-ring's because communication is more restricted in the 1-ring.

Kumar and Rao [12, page 21] analyzed the dependence of performance on problem size, communication cost and network connectivity. Their *isoefficiency function* estimates how big a problem has to become to maintain efficiency for systems with more processors. The implementation discussed later in this chapter should follow their analysis for the shared-memory model, which applies whenever there is complete connectivity (any processor can get work from any other processor). However, since the underlying architecture is distributed, the communication costs are much higher than those used in their analysis (probably even higher than for distributed MIMD machines, which have dedicated high-speed networks, and usually no interference from other users' programs).

Finkel and Manber [7] implemented DIB, a library package that can be used for distributed (general) depth-first search, without having to worry about the extra programming effort required for parallelism. DIB handles process creation, communication, work distribution, and fault tolerance. The user supplies the initial state, a successor-generating function, and a function to print the results.

In DIB, work is initially distributed among the processors. When a processor becomes idle (has nothing to do), it requests work from another processor. When a processor receives a request, it sends out work, if available. In addition, if a processor cannot find work to do, it redoes work that it has previously transferred to another processor (results for which it is waiting), in order to provide *fault tolerance* (see Section 6.7). This redundant work has low priority and is aborted when the results arrive, and suspended when non-redundant work arrives. Unlike in Kumar and Rao's program [12], each processor in DIB keeps track of where it sends work and from where it receives work, so that any results can be returned to the appropriate processor.

Finkel and Manber tested DIB on a variety of problems, using distributed systems containing up to 20 VAX 11/750 computers. For the 11-queens† problem, an exhaustive search problem, the efficiency was almost 100% (even for 16 processors) [7, page 245]. For the traveling salesman problem [13], there was a wider variation and speedup anomaly occurred sometimes. The figures for individual processors [7, page 247] (traveling salesman problem) showed that work was balanced almost perfectly: each processor spent almost exactly the same amount of time doing useful work. These results are consistent with those presented in Chapter 5. For alpha-beta, information deficiency is a problem, and the speedups were therefore lower (less than 3 for 8 processors) [7, page 248]. Finkel and Manber also discuss improvements [7, page 250], which would likely improve DIB's performance for more complicated search problems (e.g. alpha-beta).

It is important to note, however, that information deficiency does not occur in exhaustive searches such as IDA* and N-queens. In more complex searches, there are additional tradeoffs between information deficiency and communication and synchronization overheads, and thus the speedups should be lower. Chapter 6 describes extra complications present in branch-and-bound searches, for example updating bounds. In alpha-beta, if processors simply send away parts of their trees, then forget about them, information deficiency overhead increases substantially. Vornberger and Monien [26], and Feldman *et al.* [6] used this work distribution scheme to implement a parallel version of alpha-beta. To increase the effectiveness of the bounds, however, it is necessary for processors to wait until their slaves have returned results, though processors can obtain more nodes and search them while waiting. Information deficiency can be reduced, but not eliminated, by transferring a node only if at

† In the N-queens problem, the goal is to place N queens on an $N \times N$ chessboard in a position in which no two queens attack each other.

least one of its siblings has already been evaluated [6].

The fact that parallelism is initiated by idle processors helps to reduce information deficiency because it avoids the creation of excess parallel tasks. Work is parallelized, at the expense of various overheads (especially information deficiency, in some problem domains), only when processors become idle (and have nothing better to do), not when processors decide that they have too much work to do. If tasks that are intended to be done simultaneously are instead done sequentially (i.e. one task after another, rather than being merged into one task), they suffer from parallel overheads, while gaining no benefit from parallelism.

4.3. Description of Algorithm

This section describes PDFS (abbreviation for "Parallel Depth-First Search"), an implementation of a distributed depth-first search algorithm that uses the asynchronous, decentralized work distribution scheme discussed in the previous section. The technique used in PDFS is general, although the present implementation lacks some of the operations needed in complex search domains.

PDFS runs on N processors, numbered from 0 to $N-1$. Each processor runs the same program, except that the processor numbered 0, the *root* or *master* processor, also receives input data, which it sends to the others, and arranges for orderly termination when the search ends. Processors communicate over a network, which may or may not allow any particular pair of processors to communicate with each other directly.

All processors run PDFS for the duration of the search. A processor is called *busy* when it is searching nodes, and *idle* when it is not. Initially, the master processor is assigned the entire tree, and is therefore busy, and the other processors (*slave*

processors) are idle. Busy processors search as a sequential program would, except that they may be interrupted when they receive messages from other processors. Whenever a processor is idle, whether it starts out idle or becomes idle when finishing its work load, it tries to obtain work from other processors, unlike in many other algorithms, in which idle processors simply wait for work to arrive.

To obtain work, an idle processor sends a message, requesting work, to another processor. If a processor receiving this message is busy, it interrupts its searching and looks for nodes to transfer to the requesting processor. If it finds available nodes, it sends them to the requesting processor, then continues searching; otherwise, it sends a rejection message. If the processor receiving the request is idle, it immediately rejects the request. When the requesting processor receives work, it becomes busy and begins searching. If it receives a rejection message instead of work, it must try to get work elsewhere. Idle processors continue to request work from all other processors, one at a time, until work arrives or the search finishes. Each processor has a variable indicating the next processor from which it should request work. The variable is initially 0, because processor 0 is the only one that has work at the beginning, and is incremented (modulo N) every time the processor sends a request, regardless of whether it results in work. When incrementing this counter, a processor skips its own number or that of any processors with which it cannot communicate.

A unit of work consists of one or more subtrees whose roots are successors of nodes in the current search path. The appropriate number and combination of these subtrees depends on the problem domain. "Transferring a node" means transferring the subtree rooted at that node: the node and all its descendants. This method was chosen because it can be implemented easily in most search domains and it does not require large, complicated data structures — only a stack, whose additional memory requirements are linear in the search depth. In addition, PDFS includes a simple

mechanism to control the sizes and locations of transferred subtrees, since these factors can affect various overheads substantially. Only subtrees rooted at depths between two parameters, *MINDEPTH* and *MAXDEPTH*, can be transferred. Despite the apparent simplicity, the experiments (see Chapter 5) will show that, at least for simple problem domains, more elaborate work distribution methods are unnecessary.

The search ends when either the problem is solved or all processors have become idle, which means that all nodes have been searched. Except for extra operations at the beginning, synchronization at the end, and the initial allocation of the entire tree, the master processor is identical to the slaves: when it becomes idle, it looks for work, just like any other processor. This similarity among processors eliminates the possibility of one processor becoming a bottleneck in larger systems: except at the beginning, the master processor is no more likely to receive requests than is any other processor. The pseudocode in Figures 4 and 5 outlines the processors' main operations.

Because the master's extra overheads at startup and termination are constant and occur only once, regardless of the size of the problem, these extra duties should have little effect on the speedup unless too many processors are being used to solve a small problem. Alternatively, the master could do only these additional functions, while an additional slave process, running on the same machine, could do the searching. The startup overhead can be reduced further by initially partitioning the tree, assigning some nodes to each processor. After searching its initial set of nodes, a processor looks for work, as before. However, this method does not necessarily work well for some algorithms, such as alpha-beta, because it greatly increases information deficiency.

```
search1(state)     /* for depth <= MAXDEPTH */

  generate bit-mask of valid successors
  store bit-mask at top of stack
  clear bit corresponding to reversal of last move
  for each legal move m
    if (bit m of bit-mask is 1) and (problem not solved)
      clear bit m of bit-mask
      make move m
      increment node count
      if (no cutoff)
        if (depth<=MAXDEPTH)
          push m and 0    /* space for bit-mask */
          if (goal_node or search1(new_state)==SUCCESSFUL)
            pop()
            return (SUCCESSFUL)
        else
          if (goal_node or search2(new_state)==SUCCESSFUL)
            pop()
            return (SUCCESSFUL)
      else
        if there are any messages waiting
          clear message_flag
          poll()

      undo last move (m)
  pop()
  return (UNSUCCESSFUL)
end

search2(state)     /* for depth > MAXDEPTH */

  for each legal move m (not including reversing last move)
    make move m
    increment node count
    if (no cutoff)
      if (goal_node or search2(new_state)==SUCCESSFUL)
        store last move
        return (SUCCESSFUL)
    else
      if there are any messages waiting
        clear message_flag
        poll()

    undo last move (m)
  return (UNSUCCESSFUL)
end
```

Figure 4: Searching functions (all processors)

```
main()
  initialize stack
  done=FALSE

  if (master)
     create processes on other machines
     input initial state
     send initial state and other data to other processors
     indicate that master has white token
     idle=FALSE
  else
     receive initial state and other data from master
     idle=TRUE

  while (not done)
     if (idle)
        if processor has token, pass it on
        look for work
     else
        set current state to initial state
        make moves in stack to create new current state
        search1(current_state)

  /* all processors are now idle */

  if (master)
     send termination message to other processors
     receive any messages still in transit
  else
     send termination message to other processors
     receive any messages still in transit

  print results
end
```

Figure 5: Main program (all processors)

The main program (see Figure 5) is first run by processor 0, which arranges for the creation of slaves on other processors. Each processor's stack is initially emptied. Processor 0 obtains work (the input data) from the user, then initializes its stack so that it corresponds to just the root node. The main `while` loop, which runs for the duration of the search, consists of looking for work (if the processor is idle), searching the subtree it received, and passing on the token (see Section 4.4.5) if necessary.

The loop terminates when the search is finished, after which the processors synchronize (to allow for an orderly exit), then print statistics and exit. In some search domains, it is necessary to combine the results of each processor's search, after which processor 0 can output this information; otherwise each processor can output its own results.

Within the main loop, `search1` (see Figure 4) is called to search the subtree. `search1` searches the subtree, but also maintains a bit-mask to check for, and avoid searching, successors that have been transferred to other processors. When the recursion continues to depth *MAXDEPTH*, control switches to `search2`, which has fewer overheads because it is no longer necessary to check or update the stack.

Whenever reaching a leaf node, a processor checks a flag to see if any messages are waiting. (This flag is set by an interrupt handler, which is activated whenever a message arrives.) If messages are waiting, the processor temporarily stops searching and calls `poll` to respond to them. If a request for work is received, `poll` calls another function that looks through the stack to find unsearched successors. If an unsearched successor is found, it is sent and the stack is updated to indicate that this successor is no longer available; if not, a rejection message is sent. The work allocation function can be modified to implement a wide variety of work distribution methods, some of which may be more appropriate for more complex searches.

4.3.1. Major Overheads

The four main overheads of PDFS are communication, starvation, stack management (see Section 4.4.3) and, in some domains, information deficiency. Processors need to communicate to transfer work and to detect termination. Idle processors must send messages over a network and wait for replies. A busy processor that receives a message must interrupt its searching in order to respond in a reasonable amount of

time. If a request for work (the most frequent type of message) is received, the interrupted processor must also look for work to send out, and both processors must update their stacks.

Communication overhead can be reduced by decreasing the total number of messages, especially requests for work. Increasing the size of transferred work loads can result in fewer total requests, but can also cause processors to remain idle because work becomes harder to find. Another way to reduce the number of requests a processor receives is to restrict the connectivity, thus reducing the number of processors from which a processor can request work or receive a request for work. However, such a limitation could also increase idle time if distant processors have work to assign, but adjacent processors do not.

When a processor finishes searching its subtree, it must attempt to get work from other processors, and continue to do so until it either finds work or gets a termination message. This overhead can be reduced by allowing smaller (deeper-rooted) subtrees to be sent out: since the sending processor would have more choices of subtrees to send out, the request is less likely to be rejected. Unfortunately, sending out smaller subtrees would also result in more overall requests being sent because processors will finish searching their (smaller) subtrees sooner. For good speedups, a suitable tradeoff between synchronization and communication overheads must be found.

Many processors are idle at the beginning because only one processor has any nodes to search. This overhead is small, however, because the slave processors can get work quickly: since only the master processor initially has work available, each slave processor sends its first request to the master processor. Unless a goal node is found, which forces all processors to terminate immediately, many processors are idle at the end also. The work available from the few busy processors is too small to send out profitably because the overheads incurred in handling requests and sending work

exceed the savings provided by parallelizing the work. The experiments show that the amounts of idle time at the beginning and end are short.

In a recursive subroutine, a processor normally accesses only those local variables corresponding to the current invocation of the subroutine. However, when looking for work to distribute, a processor must examine data that corresponds to local variables in other invocations of the recursive subroutine. To allow access to stack entries for other invocations of a subroutine, it is necessary to maintain extra data structures. PDFS uses an explicit stack, which contains all information that needs to be shared. Maintaining this stack increases the node expansion cost because extra operations must be performed whenever entering and exiting the subroutine (i.e. for every node expansion); however, these overheads can be reduced substantially, as discussed in Section 4.4.7. In addition, every time it searches a node, the parallel program must check and update a bit-mask, which indicates which successors are available to be transferred.

In many parallel algorithms, processors lack information that would be available to a single processor executing a sequential algorithm. This lack of information may cause processors to do extra work that the sequential algorithm would have avoided. A common example in the domain of search is in alpha-beta: when sibling nodes are searched sequentially, the alpha and beta bounds from older siblings increase the number of cutoffs in younger siblings. On the other hand, when some sibling nodes are searched simultaneously, the older ones cannot cause the bounds for the younger ones to be tightened. The lack of information (tighter alpha and beta bounds) causes the parallel algorithm to miss cutoffs and thus search more nodes, resulting in lower speedups.

4.4. Implementation

The remainder of this chapter describes some of the issues related to the implementation of PDFS.

4.4.1. Work Distribution

Deciding which nodes to transfer when distributing work is an important issue because poor choices can increase overheads. As mentioned earlier, PDFS distributes work by reassigning one or more subtrees rooted at unsearched successors of a nodes in the search path; however, the experiments in Chapter 5 deal only with single subtrees. The size of this subtree affects the efficiency of the parallelization. If the subtree is too small, the requesting processor will quickly finish searching it, then request more work, resulting in extra communication overhead. If the subtree is too big, the sending processor would give away most of its nodes, thus soon becoming idle and having to request work itself, as well being unable to fulfill requests that it may receive before more work arrives. In other words, the time required for each processor to search its work load should be large compared to the time required for communication.

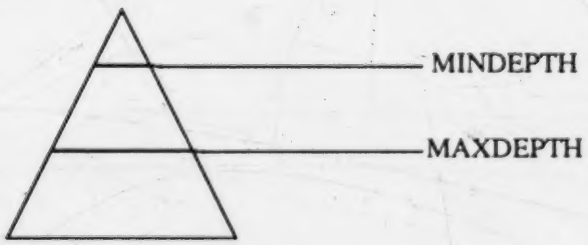


Figure 6: Choices of nodes to transfer

In PDFS, a subtree can be transferred only if its depth is greater than *MIN-*

DEPTH but not greater than *MAXDEPTH* (i.e. the subtree's root node must be located between the lines in Figure 6). There are usually several subtrees, rooted at various depths, that meet this condition. A subtree rooted closest to the root should be chosen because it is likely to be bigger than a deeper-rooted subtree. PDFS arbitrarily chooses the leftmost of these shallowest-rooted subtrees for transferral, if there is more than one.

One aim of this research is to provide insight into how to select suitable values for *MINDEPTH* and *MAXDEPTH*. There are several factors to consider when choosing these values. If *MAXDEPTH* is large, smaller subtrees can be transferred. This freedom reduces idle time because a processor receiving a request for work is more likely to find subtrees to send out, since increasing *MAXDEPTH* enlarges the set of nodes that can be transferred. However, a large value of *MAXDEPTH* also increases total communication overhead because smaller subtrees are searched more quickly than larger ones: processors become idle quickly, resulting in more frequent requests and work transfers. In addition, since deeper-rooted subtrees can be transferred, extra stack operations (see Section 4.4.7), which increase the cost of node expansion, are required for more nodes. The number of nodes for which this extra overhead is necessary increases exponentially with respect to *MAXDEPTH*, until cutoffs start reducing the number of nodes per level.

Conversely, if *MAXDEPTH* is small, work that could otherwise be parallelized is done sequentially while available processors remain idle for long periods. After all nodes rooted at depths less than or equal to *MAXDEPTH* have been searched, busy processors, which may be searching large subtrees, cannot send any nodes to idle processors. Idle processors do nothing except issue requests, which will always be rejected because no work will ever become available. In addition, these requests slow down the busy processors, which must interrupt their searching, only to find that no

work is available. If the work loads are highly unbalanced, one processor may search a large subtree for a long time, while all others remain idle.

If *MINDEPTH* is small, a processor could send away most of its tree, and thus soon become idle. If *MINDEPTH* is large, all subtrees sent out are forced to be rooted deeper in the tree, thus tending to be smaller, resulting in more communication overhead (messages and interruptions), as in the case with large *MAXDEPTH* values. In addition, idle time accumulates, even though work is obtained quickly.

The experiments, described in Chapter 5, show that the risk of transferring too many nodes (when *MINDEPTH* is small) is less important than the extra communication overhead when *MINDEPTH* is large, hence small *MINDEPTH* values are preferable. However, large *MINDEPTH* values could be useful in centralized systems, in which only one processor allocates work, to avoid the possibility that the master processor runs out of work to send. Also, in more complex searches, such as alpha-beta, information deficiency must also be considered. Larger *MINDEPTH* values would be preferable in this case also, since information deficiency is easier to control in smaller subtrees. Alternatively, the subtrees farthest from the root (instead of the ones nearest the root) can be transferred first.

It should also be noted that this scheme (using *MINDEPTH* and *MAXDEPTH* to regulate subtree size) is only approximate. The sizes of subtrees rooted at the same depth can vary greatly because of variations in the tree structure (e.g. number of successors). These variations are often difficult to predict in advance.

As often happens in parallel algorithms, there is a tradeoff between overheads: a change (e.g. in *MINDEPTH* and *MAXDEPTH*) that reduces one overhead often increases another. To obtain good speedups, it is necessary to balance these overheads. As the experiments will show, reasonable values of *MINDEPTH* and *MAX-*

DEPTH will yield almost linear speedups in some problem domains, despite the simple method of work distribution.

4.4.2. Other Work Distribution Strategies

Kumar and Rao [12, page 8] tried to split the tree into approximately equal-sized pieces by reassigning half of the unsearched successors of each node in the search path ("splitting the tree down the middle"). In the experiments described in this thesis, PDFS sends out only a single subtree, although PDFS allows several subtrees to be transferred at once (including Kumar and Rao's splitting method) without increasing communication overhead, since the entire stack is sent in either case. However, all the subtrees that are transferred must satisfy the *MINDEPTH* /*MAXDEPTH* restrictions previously described.

Although the work loads resulting from (almost) equal splitting are normally more balanced than for single-subtree transferral, there should not be a great difference if good values of *MINDEPTH* and *MAXDEPTH* are chosen. With either work distribution strategy, each processor's resulting work load is large compared to the cost of transferring work. Also, requests from other processors can increase or decrease an imbalance, regardless of how nodes are distributed. Even if processor A gives exactly half of its work to processor B, it is likely that one of these processors will receive more requests than the other because all processors independently decide from where they will attempt to obtain work. In practice, it is hard to predict when a split will be even: the numbers of cutoffs in the subtrees often differ substantially, and the extent to which the current tree has been searched can vary greatly.

For some types of search, especially alpha-beta, the choice of nodes to send is important for reducing excess search caused by information deficiency. Some extra restrictions are discussed in Chapter 6. Since the code that determines which nodes,

if any, are transferred is confined to one subroutine, it is easy to change the work distribution scheme. For example, in principal variation splitting [15] and other parallel alpha-beta algorithms, it may be better to transfer deeper-rooted subtrees first, to reduce the information deficiency overhead. Making *MINDEPTH* and *MAXDEPTH* variable might sometimes be helpful.

4.4.3. Details of Work Distribution

As mentioned before, a unit of work consists of one or more subtrees rooted at successors of nodes in the search path. Figure 7 shows an example tree, in which one processor is about to send three subtrees, rooted at the nodes marked A, B and C, to another processor. Figure 8 shows the other processor's tree after receiving the nodes. In both diagrams, the squares represent nodes in the current search path, the crossed-out circles represent nodes (roots of subtrees) that have been searched or have been allocated to another processor, and the other circles represent nodes (roots of subtrees) that still need to be searched. In this example, each node has at most four successors; the lines that do not lead to circles represent invalid operations (such as retracting the move just made in the 15-puzzle, or exposing a king to check in chess). After transferring nodes A, B and C, the sending processor would mark them as unavailable (crossed-out circles).

To access information about all nodes in its current search path, each processor maintains an explicit stack, whose entries correspond to nodes at depths from 0 to *MAXDEPTH*. (The size of each processor's stack is linear in *MAXDEPTH*.) No entries are needed for depths greater than *MAXDEPTH* because they cannot be transferred. An ordinary stack, like the one used in the sequential algorithm, can be used, thus eliminating the extra stack overhead for these nodes, which make up most of the tree.

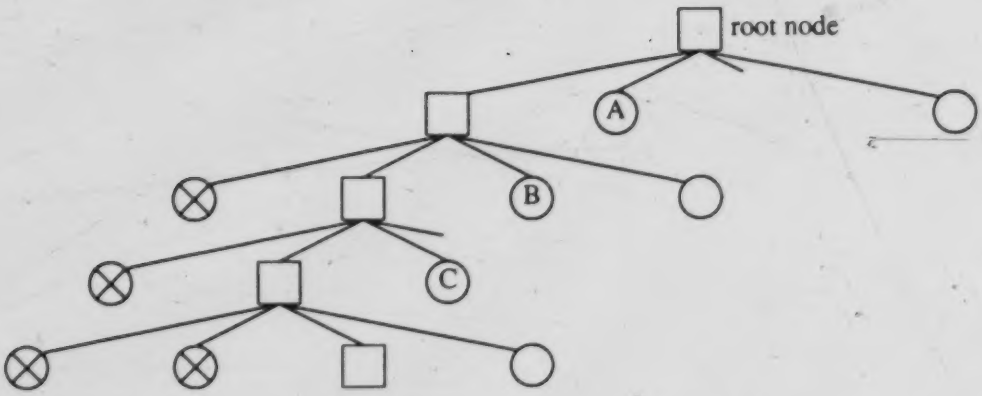


Figure 7: Search tree of sending processor (before transferring nodes)

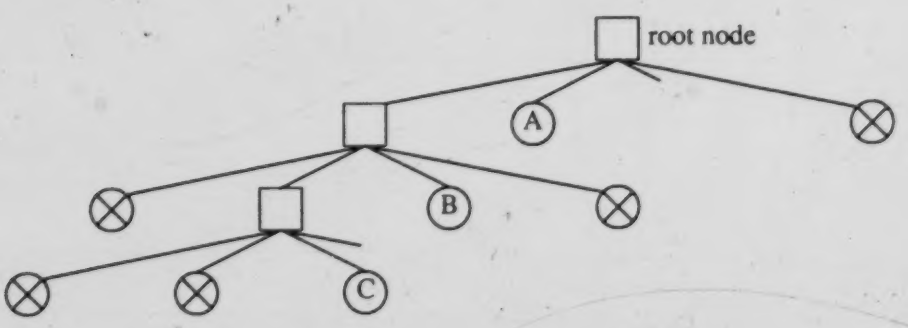


Figure 8: Search tree of receiving processor (after receiving nodes)

Figure 9 shows three stacks, which correspond to the work transfer shown in Figures 7 and 8. Every row in each stack represents a stack entry. The entry in row k, as labeled by the numbers on the left, corresponds to the node at depth k in the search path. Each stack entry contains two fields: the last move made and a bit-mask corresponding to valid successor nodes. The last-move field (leftmost column) indicates which branch was taken in order to reach the corresponding node from its parent node: it enables processors to reconstruct the current state from the initial state by making the appropriate sequence of transitions, or to output a complete solution.

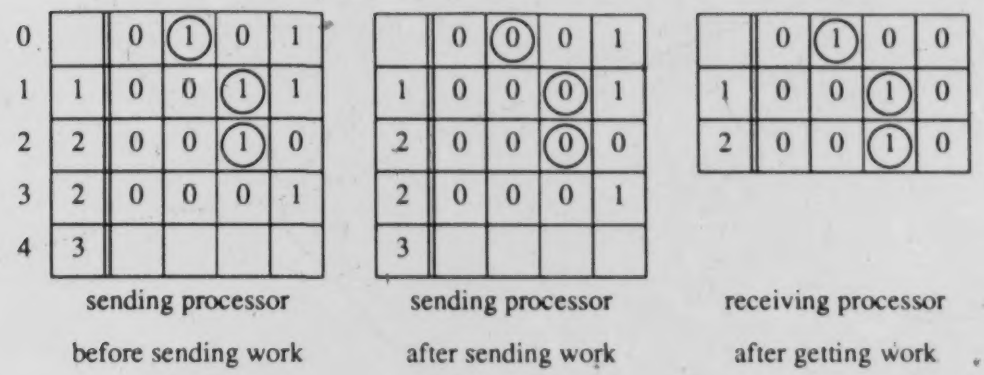


Figure 9: Stacks before and after work transfer

path if a goal node is found. (The last move field for depth 0, the root node, is irrelevant.) The bit-mask (all columns except the leftmost) indicates whether the corresponding successor is available (for search or transferral) to the processor owning the stack. A zero entry indicates that one of the following is true:

1. The corresponding move cannot legally be made.
2. The corresponding successor has been sent to another processor.
3. The processor is currently searching a descendant of the corresponding successor.

Whenever a successor of a node whose depth is less than *MAXDEPTH* (a node that is represented by a stack entry) is about to be searched or transferred, its bit-mask entry is cleared, thus indicating that the node is no longer available. Before a processor searches such a node, it checks the corresponding bit-mask to ensure that the node is not being searched elsewhere.

When work is transferred, the sending processor copies its stack, then updates the bit-masks in each copy so that they indicate which nodes are now available to

each processor. In the original stack, the bits corresponding to the root of each subtree sent are cleared, thus indicating that they are no longer available to the busy processor, as shown by the left and middle stacks in Figure 9. In the duplicate stack, the bits corresponding to these subtrees are set, thus indicating that these subtrees need to be searched, and all other bits are cleared, as shown in the rightmost stack in Figure 9. This scheme guarantees that each valid successor is searched exactly once. The duplicate stack is then sent to the idle processor, which uses it as its new stack. In Figure 9, the bits representing transferred nodes are circled. The idle processor then uses the last-move information to obtain its new current state from the initial state, which it received when PDFS started, then begins searching. (Note: the depth of the deepest node sent out is also transferred, so that the idle processor knows when the current state has been reached.)

In a shared-memory system, in which each processor can access each stack, the idle processor does the copying. Mutual exclusion is required when modifying the busy processor's stack, but the busy processor is not otherwise delayed. In a distributed system, however, one processor cannot access another processor's stack. The busy processor must suspend its own searching to look for work and, if work is found, copy the stack, make the appropriate modifications, and send the new stack to the idle processor. Thus the communication cost is more than just the cost of sending the message.

In some applications (e.g. the 15-puzzle, which is used as an example in this thesis), the time spent accessing the stack is a significant fraction of the total time spent expanding a node. In such applications, it is important to eliminate the stack management for as many nodes as possible because it increases the amount of time required to search *each node* (in the appropriate range of depths). To reduce this overhead, two slightly different subroutines are implemented. The subroutine used

for nodes shallower than *MAXDEPTH* (`search1` in Figure 4) includes extra operations, such as checking and modifying the bit-mask for every node to ensure that subtrees are never searched more than once. The subroutine for deep nodes (`search2` in Figure 4) is almost identical to the sequential algorithm (it does not access the explicit stack), except that it checks for messages from other processors. These messages could indicate that processors are looking for work or that the problem has been solved. Implementing two subroutines is better than simply testing for $depth \leq MAXDEPTH$ because it avoids the extra test, even though the cost of this test is small.

The stack structure allows several disjoint subtrees to be sent to one processor in one message, but the root of each subtree must still be a successor of a node in the search path, and therefore represented by a stack entry. However, only one subtree is transferred at a time in the experiments described in this thesis.

4.4.4. Interrupts

When a busy processor receives a message, usually a request for work, an interrupt occurs. The interrupt handler then sets a flag, which the processor checks periodically. If the flag is set, the processor will reset it, then receive all waiting messages and respond to them appropriately. (Idle processors are continually polling while waiting for replies to their requests; they do not need to check this flag.) The interrupt flag must be checked frequently to avoid keeping processors waiting, and thus not doing any useful work, for long periods. It could be checked during each node expansion, which would increase the cost of the expansion by a small amount. To reduce some of this overhead while maintaining frequent polling, the flag is checked only during the expansion of leaf nodes (see Figure 4). A method for eliminating the extra cost is discussed in Chapter 6.

4.4.5. Termination Detection

To avoid wasting potential computing power, all processors should stop at approximately the same time. If a processor terminates earlier than the others, its CPU time is wasted. Termination detection is easy when a processor has solved the problem because all processors can simply be told to stop.

However, if there is no such event that signals completion (for example, in branch-and-bound algorithms and in non-final iterations of IDA*), termination occurs only when no more nodes need to be searched. It is therefore necessary to determine when all processors are idle (i.e. no more work is available). Unfortunately, termination detection is not as trivial as it might seem. Simply polling all processors, asking them if they are busy, does not work. Consider the following possibility:

Processor A wants to check if processor B and C are finished.

Processor A polls processor B.

Processor B replies that it is idle.

Processor B asks for, and receives, work from processor C.

Processor C finishes its remaining work.

Processor A polls processor C, which replies that it is idle.

A concludes that B and C are both idle, but B is not.

One way to avoid this problem would be for B not to request work until it receives another message saying that the conjecture that the search is finished is incorrect. Another would be for B to poll other processors as well. These, and similar methods are undesirable for two resons:

1. Processor A must wait until B and C (and many others for larger configurations) have replied to the poll, resulting in wasted computing resources.

2. Many messages ($O(N^2)$ for some methods) must be sent.

The excessive messages and wait times become more and more burdensome as the system becomes larger.

One method that works if all processors must return results to the processor from which they obtained work, is for processors to say that they are idle only if they are not waiting for results from other processors (and, of course, not working themselves). However, in this thesis, processors do not have to wait for others, so this method of termination detection will not work.

The termination detection method used in this thesis is the one by Dijkstra *et al.* [4], in which a "token" is sent around a ring of processors, numbered from 0 to $N-1$, in descending order, starting with processor 0. The "ring" is simply a predetermined path that allows the token to visit all the processors. The token and each processor is considered to have two states, called "white" and "black" by Dijkstra *et al.* When the token is first sent, it is considered to be white, as are all processors. A processor that sends work to a higher-numbered processor, which may have thought that all processors were finished, becomes black. A black processor always turns the token black before passing it on; a white processor leaves the token color unchanged. A processor becomes white immediately after passing the token. When processor 0 receives a white token, all processors have finished searching. The token may traverse the ring several times before all processors finish.

Passing the token constitutes an additional overhead: N extra messages, one for each processor, for each traversal of the ring of N processors, plus an interruption and several (few) extra instructions that decide what to do with the token. However, this overhead is small because, although the token may be received at any time, it is not passed on until the processor that holds it becomes idle, which is usually a long time.

Experiments show that the token makes few traversals before termination is detected.

4.4.6. Deadlock Prevention

In a parallel program where work requests can be sent to any processor at any time, there is a danger of deadlock. Deadlock occurs if there is a subset of processors in which every processor waits for a reply from another processor in the subset. The simplest example is when two idle processors request work from each other simultaneously and wait for replies from each other. To prevent deadlock, when processor A requests work from processor B, which may itself be waiting for work, A should get an immediate reply so that it can look elsewhere for work. Otherwise, A would have to wait until B receives a reply: if B is waiting for A, deadlock would arise, since neither processor can reply to the other until it receives a reply.

4.4.7. Effects of Increased Node Expansion Cost

If the amount of time required for the parallel algorithm to expand a node exceeds the time required for the sequential version, the speedups are reduced. For this analysis, it is assumed that the tree is uniform with depth D and branching factor b. It is also assumed that both the sequential and parallel algorithms search every node in the tree. The analysis concentrates on the overheads of stack management. The analysis for the interrupt flag is similar, but the extra overhead is smaller and depends on the number of leaf nodes.

Let M=*MAXDEPTH*.

Let $n = \sum_{d=0}^{D} b^d = \frac{b^{D+1}-1}{b-1}$ be the total number of nodes searched.

Let $k = \sum_{d=0}^{M} b^d = \frac{b^{M+1}-1}{b-1}$ be the number of nodes for which stack management

is necessary (depth is less than *MAXDEPTH*).

Let N be the number of processors.

Let α be the cost of searching a node.

Let ε be the extra cost (for each node expansion) of managing the stack.

The sequential time is αn. The parallel time is at least $\frac{\alpha n + \varepsilon k}{N}$, resulting in a maximum possible speedup of $N\frac{\alpha n}{\alpha n + \varepsilon k} = \frac{N}{1+(\varepsilon k)/(\alpha n)}$. If $k=0$, the maximum possible speedup remains N. At the other extreme, if $k=n$, the speedup is limited to $\frac{N}{1+\varepsilon/\alpha}$. For problems in which the node expansion cost (α) is low, ε might not be significantly less than α. If $\varepsilon=\alpha$ (in addition to $k=n$), then the maximum possible speedup is only $\frac{N}{2}$.

Unless d is close to D, k is much smaller than n, and therefore stack management overhead is insignificant. If b is large and ε is much smaller than α, stack management may be insignificant even if $MAXDEPTH = D-1$.

There are additional considerations to take into account for some problem domains. For instance, in the example in Chapter 5, there are cutoffs in the deeper parts of the tree, which cause the number of nodes per level to stop increasing exponentially and eventually start decreasing. In this case, increasing *MAXDEPTH* beyond a certain point has little effect.

CHAPTER 5

5. Experiments

To measure the efficiency of the decentralized parallel depth-first search algorithm described in the previous chapter, PDFS was implemented in a distributed environment. PDFS is written in C and was tested using Sun 4 workstations, all of which are running the UNIX operating system and are connected to an Ethernet network, which allows any pair of machines to communicate with each other. PDFS had one process running on each machine.

Processes communicate using the Network Multi-Processor (NMP) [16][2] package. NMP, developed at the University of Alberta, consists of a set of subroutines that provide a simple interface to the UNIX socket handling routines, as well as a server that creates processes on remote machines. NMP can simulate systems with arbitrary connection topologies, although complete connectivity was used in these experiments. The implications of reduced connectivity are discussed in Section 6.1.

The search domain is a simple heuristic search that attempts to find a solution to a problem, given its depth and an admissible heuristic. PDFS searches all nodes whose cost plus heuristic evaluation is less than or equal to a fixed bound. To avoid the possibility of speedup anomaly, a bound that is less than the known solution depth was used, thus forcing the parallel program to search the same set of nodes as the sequential one. The largest such bound was used in order to allow experimentation with large trees.

The possibility of speedup anomaly was eliminated because it is non-deterministic: the number of nodes searched before a solution is found depends on the order in which successors are searched. The node count would change every time

PDFS is run, even with the same input data, because unrelated factors, such as network traffic or other processes, may affect the way that nodes are transferred. (In a sequential algorithm, successor ordering is deterministic; the node count changes only if the successor ordering algorithm is changed.) The effects of speedup anomaly greatly overwhelm those of the communication and synchronization overheads. The resulting confusion over the cause of the timing differences complicates the analysis of PDFS's efficiency.

An alternative to avoiding solutions (which would not be realistic for actual applications anyway) would be to run the program several times and use the average time, or to compare the total time divided by the number of nodes searched. However, the results can be reproduced more easily if the source of the uncertainty (solutions) is eliminated.

Since the purpose of this thesis is to investigate parallel depth-first search, not to solve a particular problem, PDFS was designed to be general enough to allow easy modification to take into account the differences in particular algorithms, such as iterative-deepening, changing bounds, or two-player searches. Implementing one particular search algorithm (e.g IDA*, branch-and-bound, alpha-beta), with all its specific features, does not help when trying to generalize the results. With some modifications, discussed in Section 6.5, PDFS can be used to solve a wide variety of more specific, complex problems.

5.1. Problem Domain

The problem domain in these experiments is the 15-puzzle, a special case of the $m \times n$ puzzle. It consists of a 4×4 board with tiles numbered from 1 through 15 occupying fifteen of the squares. A *move* consists of sliding an adjacent tile vertically or horizontally onto the blank square. The goal is to reach the final state in as few

moves as possible. There are initial positions from which the problem cannot be solved (i.e. the graph that represents the 15-puzzle is disconnected); these positions are not used. Figure 10 shows the goal state and the initial states of the four problems used for the experiments in this thesis, in increasing order of difficulty. A problem's number is the order in which it appears in Korf's paper [10]. Problems 66, 60, 82 and 88 were the four most difficult (most nodes searched by IDA*) examples that Korf used [10]. For the search algorithm used in PDFS, problem #66 was the fifth most difficult, but the other three retained their rankings of difficulty.

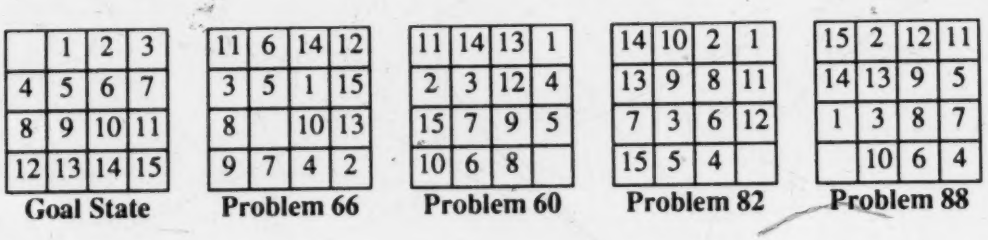


Figure 10: The 15-puzzle

IDA* is an effective algorithm for solving the 15-puzzle (A* requires too much memory). The most commonly used heuristic, which is also used in this thesis, is the *Manhattan distance*. Mathematically, the Manhattan distance between two coordinates (x_1,y_1) and (x_2,y_2) is $|x_2-x_1|+|y_2-y_1|$. In the 15-puzzle, this distance is equal to the number of moves that would be required to get a tile from one square to the other, assuming there are no intervening tiles. The heuristic value h is the sum of the Manhattan distances of the current position of each tile to its goal square. Clearly, this heuristic is admissible because the presence of intervening tiles can only increase the number of moves required to get a tile into the correct position. The cost, g (see Section 2.5), is the number of moves already made.

The 15-puzzle is not necessarily ideal for testing all algorithms: the search tree

has a low branching factor and the cost of expanding nodes is very low. Also, the heuristic value of a node is always one more or one less than that of its parent (thus IDA*'s next threshold is always two more than the current one). The low branching factor should not be a problem for PDFS because subtrees that are sent out can be rooted at various depths. In a shallow, bushy tree, there are many nodes in a small range of depths (*MAXDEPTH*−*MINDEPTH* is small). In a deep, narrow tree, where there are proportionately fewer nodes at individual depths, a large value of *MAXDEPTH*−*MINDEPTH* allows nodes in a greater range of depths to be sent out. So, in either case, there is plenty of work to send out. Small differences in the depths of reassigned subtrees are overshadowed by variations in the number of cutoffs (threshold being exceeded) among subtrees.

The low node expansion cost is not a problem either because *MAXDEPTH* is used to eliminate the overheads for most of the nodes. Otherwise, PDFS should be much more efficient for problems with higher node expansion costs because any extra overheads would constitute a smaller proportion of the node expansion cost (i.e. the ratio of communication to node expansion cost would decrease).

5.2. Expectations

Kumar and Rao's results [12] suggest that the biggest overhead in their algorithm is communication (the times required to send requests for work and wait for replies, and, at least in PDFS, the time required for interrupted processors to look for work to send out) because their program's speedups decrease as communication costs increases. In both their program and PDFS, there is plenty of work available to transfer, except at the end of the search. Unless unnecessary restrictions are placed on the choice of subtrees to transfer, or processors are slow to respond to requests for work, no processor should be idle for long periods. The startup overhead, as well as

the idle time at the end, depends only on the number of processors, not on the size of the problem, and therefore becomes insignificant as the problem size increases.

In PDFS, stack management occurs only in the shallow levels of the tree, which contain only a small fraction of the total nodes. Passing the token for termination detection (see Section 4.4.5) does not take much time: in the experiments described in Section 5.5, the token never made more than four circuits around the ring of processors when suitable values of *MAXDEPTH* and *MINDEPTH* were chosen. Synchronization overhead is also small because it occurs only when processors wait for replies to messages, but processors spend little time looking for work. In more complicated problem domains, however, reducing synchronization overhead may result in unnecessary search caused by information deficiency.

5.3. Difficulties

There are several difficulties that complicate timing and reduce its accuracy. The main problem is the presence of processes not belonging to PDFS, since other users may start applications on any machine at any time. System processes, which may consume more CPU time on some machines than on others, are also present. Because of the way that NMP calculates times, high idle time figures often indicate the presence of such unrelated processes, rather than the amount of time that a machine is truly idle. Experiments whose measurements were corrupted significantly by such processes were redone. Corruption is assumed if the figures for individual processors differ substantially and repeated attempts eliminate the differences. Experiments in which machines were monitored for activity identified the effects of these other processes. True idle time is low when reasonable values for *MINDEPTH* and *MAXDEPTH* are chosen. Other processors affect the user time figures also: one other CPU-intensive process would consume half of the user time; system time would

also increase because of context switching. In this section, corruption is considered to be unacceptable if its effects obscure those of changes in *MINDEPTH* and *MAXDEPTH*.

In addition, the UNIX timing routines are not completely accurate. Differences in the real and user time measurements are often greater than 4%, even on identical machines running the same (sequential) program with the same input data, without interference from users' applications. The differences between consecutive runs on the same machine were much smaller, however: the difference was usually less that 0.1%, but was sometimes close to 0.5%.

These difficulties do not reduce PDFS's efficiency: idle machines search more nodes while busy machines execute unrelated processes and faster machines search more nodes in the same amount of time; the transfer of work achieves load balancing. However, because these difficulties complicate timing, only identical machines were used in these experiments, and measurements corrupted by the existence of unrelated processes were discarded and the experiment redone.

Because of these timing differences, it is difficult to obtain precise speedup figures. The choice of machine to use for the sequential algorithm can affect the speedup significantly. For this reason, the sequential program was run several times on different machines. Separate speedup calculations were made, using the fastest and slowest sequential timings from among those that were not corrupted by the presence of unrelated processes, to obtain upper and lower bounds for the speedup.

Finally, gaining access to enough idle processors was a problem. The main advantage of decentralized algorithms (over centralized ones) is that they are more effective in systems with many processors. Ideally, PDFS should have been tested in a system with hundreds of processors; unfortunately, no more than sixteen identical

processors were available. In practice, the processors do not have to be identical; however, accurate timings were desired in these experiments.

5.4. Results

As described earlier, there is a tradeoff between overheads: idle time vs. communication and stack management. To find optimal values for *MINDEPTH* and *MAXDEPTH*, PDFS was run many times, using different values for these parameters. The results of these experiments are presented in the following tables and graphs. PDFS was run on a system of 16 Sun 4 workstations (the maximum number available) of identical speed. Machines running other users' programs were avoided, although there was some interference from short programs that started during the execution of PDFS. Two of the problems in Korf's set [10] were used as input: the biggest (#88) and the fourth-biggest (for full IDA*) (#66). (A "bigger" problem is one that requires more nodes to be searched.) These problems were chosen because the purpose of parallelism is to solve *large* problems: in small problems, startup and termination overheads constitute a large fraction of the running time. Also, since there is a large difference in the number of nodes (924,074,079 nodes for problem 66 and 5,156,184,395 nodes for problem 88 — a ratio of more than 5.5 to 1), the effects of problem size on efficiency could be investigated. (The node count is the same as for the second-to-last iteration of IDA*.) The difference in the sizes of the problems may also affect the appropriate choices of *MINDEPTH* and *MAXDEPTH*.

Table 1: Average times for problem 66

Problem 66, 16 processors, 924074079 nodes								
depth limits		average times (seconds)					work transfers	
MIN	MAX	real	user	system	idle	wait	successes	failures
0	4	1023	410	576.8	35.3	649.6	32	352010
0	8	397	336	34.8	25.4	39.4	118	23167
0	12	369	333	10.9	24.2	13.1	176	7256
0	16	360	334	2.0	24.1	4.4	394	1100
0	20	382	344	1.2	34.8	4.2	444	523
0	24	405	381	1.6	21.2	5.0	501	480
0	32	545	512	1.8	29.8	5.3	509	539
0	40	587	561	2.2	22.5	6.2	549	737
0	48	597	565	2.0	28.9	6.6	649	501
0	56	628	566	5.6	55.8	6.1	501	509
4	8	744	378	333.8	31.5	375.4	559	223082
4	12	398	336	21.8	39.1	36.8	1837	13654
4	16	387	336	8.3	42.2	29.3	2824	2712
4	20	407	346	6.7	53.2	30.1	3167	784
4	24	443	384	7.1	50.8	32.7	3389	830
4	32	568	516	5.2	46.1	27.6	3087	478
4	40	625	564	7.1	51.9	32.9	3476	674
4	48	630	568	6.2	54.3	30.5	3360	513
4	56	630	568	7.8	52.6	30.6	3240	685
8	12	754	377	296.0	80.2	387.1	8215	169679
8	16	649	352	113.7	182.8	280.4	20074	23014
8	20	715	364	118.1	232.5	334.8	26345	15635
8	24	764	401	118.2	243.7	352.0	28441	12211
8	32	934	534	120.2	278.5	380.9	31871	11573
8	40	992	582	121.6	287.1	384.9	32483	10678
8	48	991	587	122.0	280.4	381.3	31988	11949
8	56	989	587	120.2	280.6	386.1	32696	12093
12	16	1605	444	801.2	359.0	1227.8	48272	318682
12	20	1883	444	695.4	742.6	1488.7	92515	149932
12	24	2142	490	741.4	909.7	1705.9	111803	129379
12	32	2539	625	816.9	1095.9	1971.6	138220	127022
12	40	2655	684	830.2	1139.8	2041.8	142336	125114
12	48	2670	689	821.9	1157.6	2042.3	143485	127527
12	56	2649	687	831.8	1129.4	2025.3	140395	128846

Table 2. Average times for problem 88

Problem 88, 16 processors, 5156184395 nodes								
depth limits		average times (seconds)					work transfers	
MIN	MAX	real	user	system	idle	wait	successes	failures
0	4	4738	2181	2481.0	74.9	2761.9	22	2062536
0	8	2085	1870	181.2	33.1	199.2	121	169456
0	12	1932	1856	41.0	34.2	46.3	229	32669
0	16	1921	1847	6.0	66.7	5.2	351	2229
0	20	1913	1861	2.3	48.6	2.8	329	559
0	24	1977	1939	2.3	34.9	3.4	410	446
0	32	2577	2536	2.6	36.8	4.5	515	626
0	40	3113	3070	2.7	38.9	5.7	634	500
0	48	3186	3128	3.8	53.4	5.6	613	617
0	56	3204	3144	2.5	56.8	4.4	485	405
4	8	2352	1901	400.4	49.6	435.5	338	394239
4	12	1940	1850	30.3	58.7	39.2	1292	22716
4	16	1920	1847	6.2	65.5	16.4	1938	1941
4	20	1929	1863	4.8	60.9	16.3	2046	1131
4	24	1997	1941	3.9	51.2	18.2	2417	633
4	32	2603	2544	3.6	54.2	15.1	2012	215
4	40	3170	3081	6.8	81.2	16.4	2101	309
4	48	3234	3136	7.1	89.5	17.1	2194	473
4	56	3201	3150	3.6	45.8	18.1	2198	296
8	12	2618	1931	597.1	89.2	685.9	6335	579796
8	16	2105	1859	58.6	186.0	202.4	20569	21157
8	20	2200	1881	72.2	245.6	280.9	27443	6195
8	24	2363	1977	79.2	305.8	312.9	30384	3945
8	32	2923	2569	66.4	286.6	304.5	31415	2980
8	40	3516	3120	75.1	319.8	321.8	33052	2872
8	48	3556	3176	70.5	308.4	314.6	32484	3015
8	56	3518	3175	65.8	276.5	308.0	32106	3036
12	16	4155	2068	1443.5	642.6	2155.3	81008	916403
12	20	4415	2015	951.6	1446.7	2446.1	185591	218963
12	24	4943	2104	964.1	1874.0	2879.9	236223	146042
12	32	6000	2715	1055.2	2228.5	3303.9	273739	131438
12	40	6588	3266	1013.2	2308.1	3344.7	284967	123129
12	48	6685	3320	1038.6	2325.1	3376.9	284578	122250
12	56	6681	3332	1013.6	2334.7	3369.8	286817	123533

Each row in Tables 1 and 2 displays the statistics for one run with the specified values of *MINDEPTH* and *MAXDEPTH*. The timing figures are averages over all processors. The figures for the node count and the numbers of accepted and rejected requests are totals over all processors. Real time means the total clock time, from start to finish, including wasted time. A process' user time is the time the CPU spends only on that process. System time is the time spent during system calls, for example to the UNIX `send` and `recv` functions. Idle time, which technically means the time spent doing nothing, also includes the time spent by the CPU on unrelated processes on the same machine; it is calculated by subtracting the total of user and system times from the real time. Wait time measures the total time spent looking for work, both sending requests and waiting for replies (the only case of true idle time). Wait time has some user- and system-time components because system calls and other instructions must be executed when looking for work. The figures for total real, user, system and idle times include the corresponding components of wait time. The total overhead is at least as large as the total wait time. The number of successful requests is the total number of times that work was transferred. The number of unsuccessful requests is the total number of times that processors requested work, but did not receive any.

One observation from Tables 1 and 2 is that the system and wait times increase, almost linearly, as the total number of requests increases. Clearly, each request and each reply requires a certain amount of system time. Wait time also accumulates as more requests are issued.

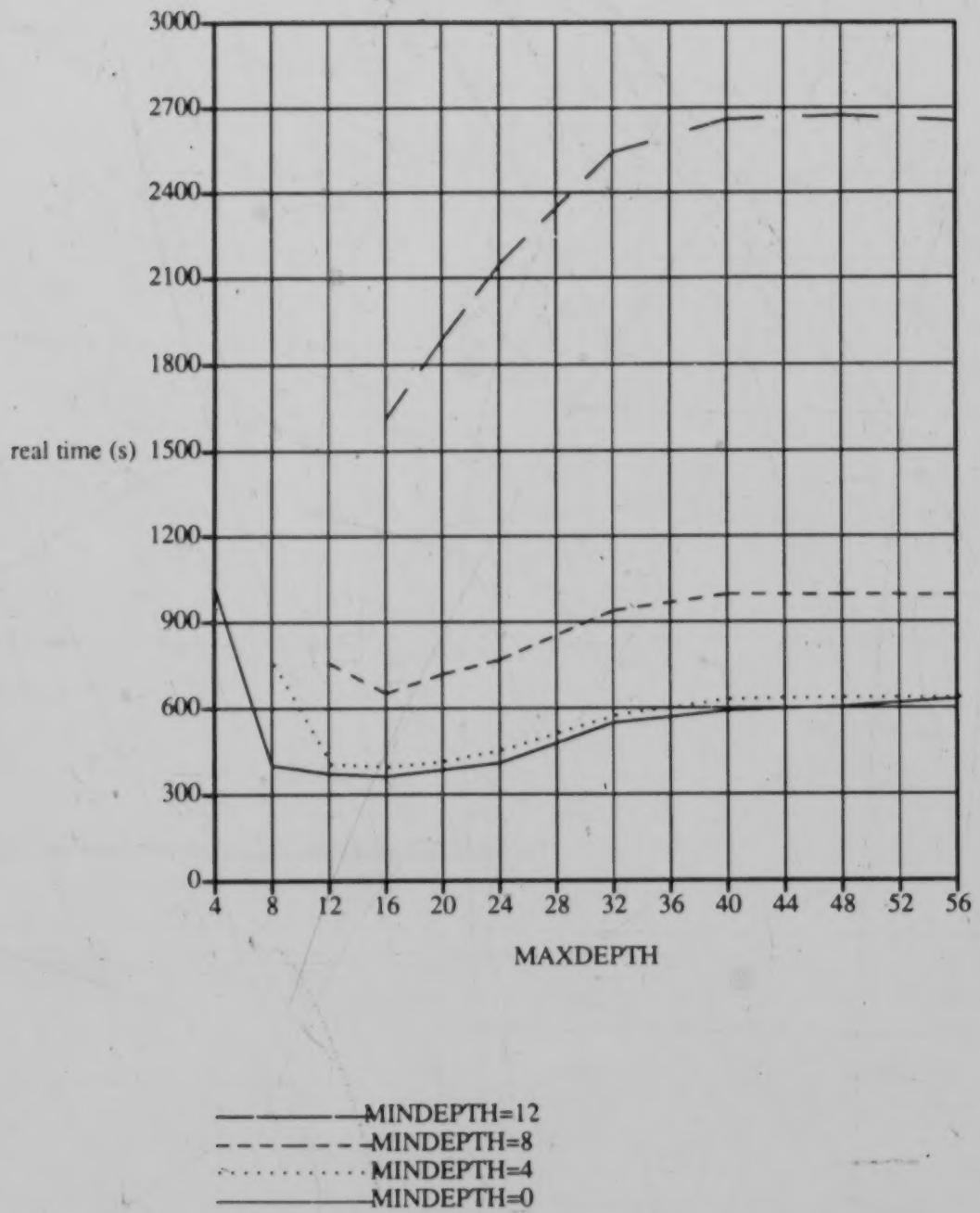


Figure 11: Time as a function of MINDEPTH and MAXDEPTH (problem 66)

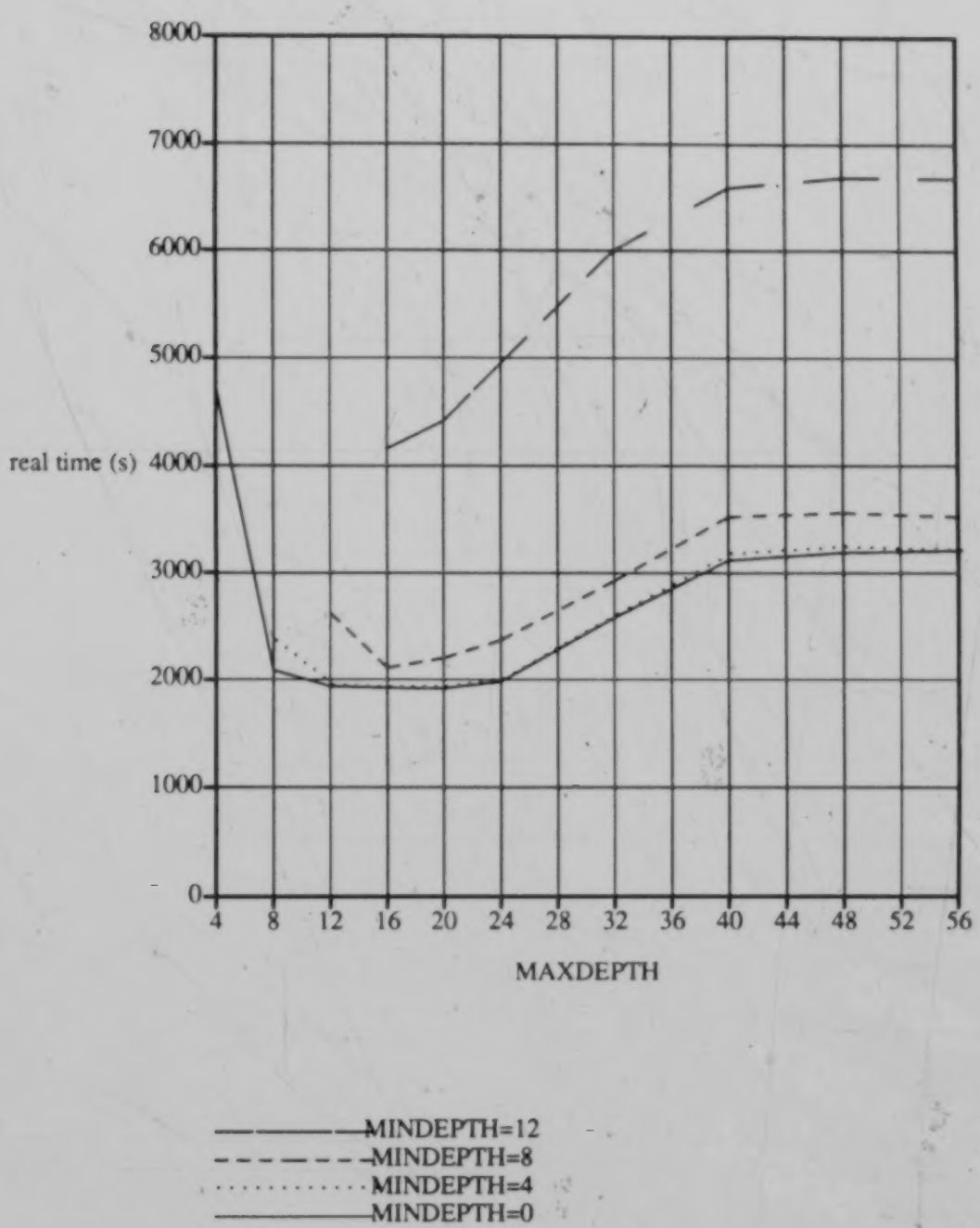


Figure 12: Time as a function of MINDEPTH and MAXDEPTH (problem 88)

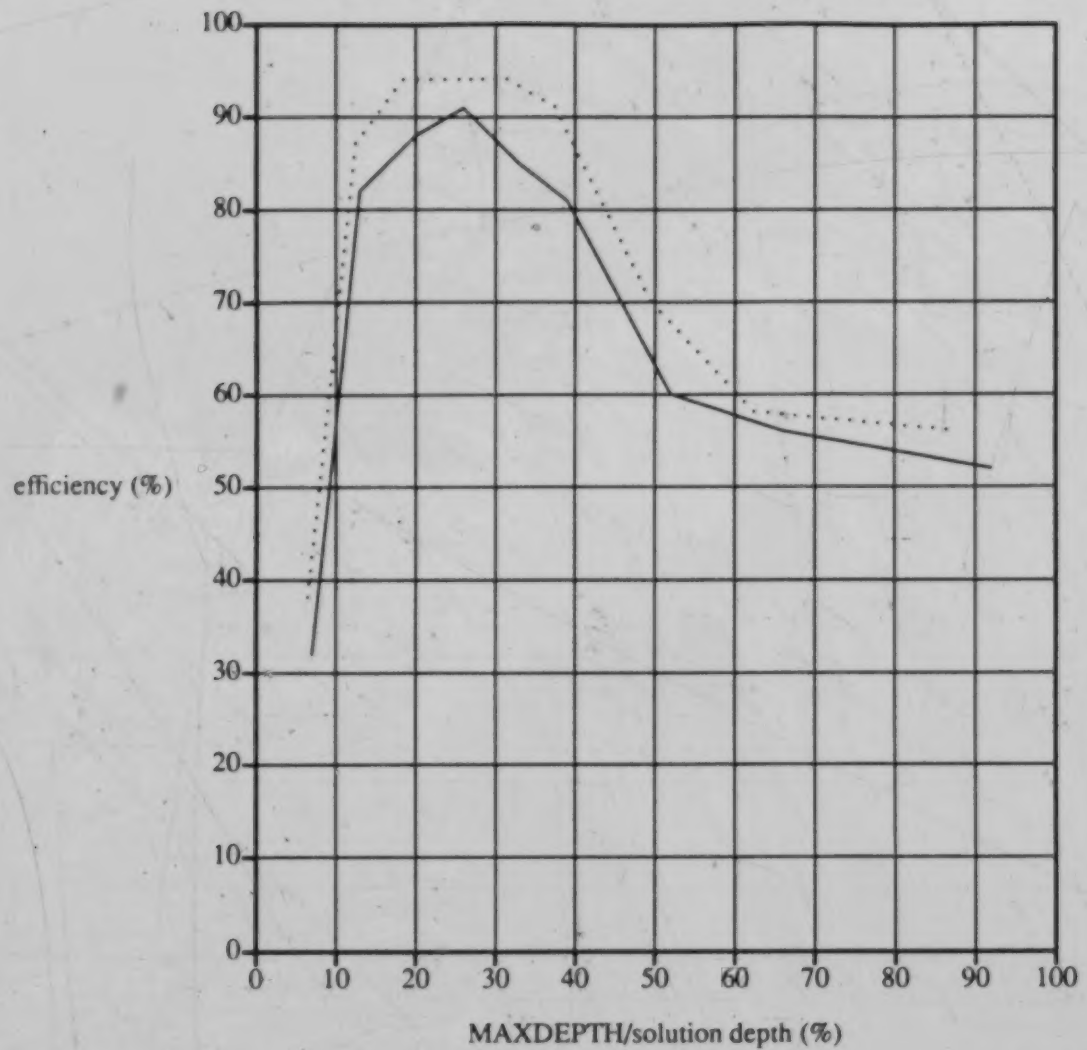


Figure 13: Efficiency as a function of MAXDEPTH and problem size

Figures 11 and 12 plot the real times in Tables 1 and 2 and show that there is a range of values for *MINDEPTH* and *MAXDEPTH* in which the amount of real time is minimal. Figure 13 normalizes the data by relating efficiency with *MAXDEPTH* (as a percentage of the solution depth) and shows that this range is approximately the same for both problems. *MAXDEPTH* was compared to the depths of the optimal solutions (61 for #66, 65 for #88), which generally increase as the problem size increases. Since the solution depth is more easily predetermined than the node count, it is a more convenient estimate of the problem size. More generally, if the solution depth is not known in advance, the maximum search depth (in this example, two less than the solution depth) can be used as an estimate of the problem size.

There is more variation within the range of optimal *MAXDEPTH* values for the smaller problem (#66), whose performance peaked at one point, rather than remaining nearly constant over the entire range. The performance for problem #88 is better than for #66 throughout the range of *MAXDEPTH* values because problem #66 is smaller: smaller units of work are distributed, but at the same cost as distributing larger units in problem #88, which means that communication overhead occupies a greater proportion of the total time.

5.4.1. Effect of MAXDEPTH

When *MAXDEPTH* fell below the optimal range, processors had trouble finding work, as shown in Tables 1 and 2 by the large number of failed attempts. The reason for this difficulty is that, since there are few nodes at shallow depths, there are few choices of subtrees to distribute. After several requests have been filled by various processors, all subtrees rooted at depths less than *MAXDEPTH* have been or are being searched, thus leaving no work available to be transferred in the future. At this point, any processor that finishes searching its subtree remains idle, doing nothing but

looking for work for the remainder of the program's execution. The idle time can be substantial, especially if the work loads were highly unbalanced. The high system time figures support the conclusion that much of the time was spent sending and replying to requests for work. This overhead increased substantially as *MAXDEPTH* decreased to near *MINDEPTH* (see Tables 1 and 2), since all available work was distributed near the start of the search.

Even as *MAXDEPTH* increased from 12 to 16 to 20, the number of unsuccessful requests dropped significantly (see Tables 1 and 2), indicating that work was still becoming easier to find. However, the increase in the number of successful requests indicated that processors had to look for work more often because they received smaller work loads. Wait time decreased along with the total number of requests.

Real time also increased as *MAXDEPTH* increased above the optimal range. One reason for this increase is the overhead of stack management: the number of nodes for which stack management is necessary increases exponentially as *MAXDEPTH* increases, until cutoffs start occurring. The increase in user time supports this conclusion, since more work is required to search individual nodes, whereas lack of work would be indicated by an increase in wait time.

This increase (in real time) started leveling off as *MAXDEPTH* increased past 32 because the cutoffs at deeper levels reduced the number of additional nodes for which stack management was required. Few nodes at depths near the threshold value had to be searched, thus for large values of *MAXDEPTH* (40 or more), the experimental error is greater than the variation in this overhead.

Tables 1 and 2 show that, for *MINDEPTH*=0, there was little change in the numbers of successful and unsuccessful requests as *MAXDEPTH* increased beyond 20. The timing differences were caused by the extra stack management required for

deeper nodes, since the increase in user time and the lack of significant change in wait time indicated that work was being done.

5.4.2. Effect of MINDEPTH

The graphs of *MAXDEPTH* vs. time (see Figures 11 and 12) have approximately the same shape for *MINDEPTH* values of 0 to 8 (except for the omission of points corresponding to *MAXDEPTH* ≤ *MINDEPTH*), but the times increase (the curves move upward) as *MINDEPTH* increases. The amount of time increases because more messages are sent: since all transferred subtrees are forced to be smaller, processors must get work more often. The wait time (see Tables 1 and 2) accumulates as more requests are issued, even though processors can find work easily. The reduction in performance is slight when *MINDEPTH* increases from 0 to 4, which indicates that the exact choice of value for *MINDEPTH* is unimportant if it is chosen to be between 0 and 4 (and perhaps slightly higher). As *MINDEPTH* increases to 8, the curve shifts upward by a larger amount: close to double for problem 66, but usually less than 15% for problem 88. The difference is smaller for the largest problem, #88, because a slightly deeper-rooted subtree is often as big as a shallower-rooted subtree in smaller problem. The results get worse as *MINDEPTH* increases to 12 and beyond.

The range of best *MAXDEPTH* values becomes smaller as *MINDEPTH* increases. This range disappears altogether for *MINDEPTH*=12, although the right side of the curve behaves the same way as for smaller *MINDEPTH* values. The overheads for *MINDEPTH*=16 were so high that continuing the experiment was pointless.

From these observations, one can conclude that the many unsuccessful requests resulting from the forced transfer of smaller units of work are much more of a burden than the risk of processors becoming idle by sending away too much work. These

results show that, at least for this simple problem domain, there is no reason to restrict the minimum depth at which work can be sent out (i.e. *MINDEPTH* should be set to zero).

5.4.3. Significance of MAXDEPTH–MINDEPTH

For all values of *MINDEPTH*, when *MAXDEPTH–MINDEPTH*=4, there are many more unsuccessful requests, and fewer successful ones, than when the difference is larger. For example, even though there are few nodes at depths less than 4, the performance for *MINDEPTH*=0 and *MAXDEPTH*=8 is better than for *MINDEPTH*=4 and *MAXDEPTH*=8. The performance differs because, when the range of *MINDEPTH* to *MAXDEPTH* is small, processors have fewer nodes that can be transferred (since only a small part of the stack can be searched), meaning that work is harder to find. This conclusion is supported by the large amount of wait time, which indicates that processors had difficulty in finding work, and the high system time figures, which indicate that much time was spent servicing the UNIX system calls (`send` and `recv`) that are issued for each message.

After a processor sends out all successors located between *MINDEPTH* and *MAXDEPTH*, it can no longer send work until it gets more work itself. This lack of work greatly increased the real time when *MINDEPTH*=0 and *MAXDEPTH*=4, but had much less of an effect for larger values of *MINDEPTH*. When *MINDEPTH* and *MAXDEPTH* both increase by the same amount, there are more nodes in the applicable range of depths, thus more requests are successful, although still only *MAXDEPTH–MINDEPTH* nodes are accessible at any one time. The increasingly poorer performance with larger values of *MINDEPTH* also helped to obscure these effects: as *MINDEPTH* increased, real time increased greatly anyway.

These effects decreased substantially as the difference *MAXDEPTH−MINDEPTH* increased. Although the number of unsuccessful requests remained large, the effects on the timings were small. When *MAXDEPTH−MINDEPTH* is larger, a busy processor can satisfy more requests before it has to seek additional work.

5.4.4. Discussion

In general, requests for work are more likely to be successful as *MAXDEPTH* increases, since the work that can be sent out for a smaller *MAXDEPTH* is a subset of that which can be sent out for a larger *MAXDEPTH*. The decrease in wait time (with increasing *MAXDEPTH*) also supports this conclusion. Unsuccessful requests result in rejections and additional requests, which interrupt busy processors and waste their time if they do not have work to send. In addition, each unsuccessful attempt to get work increases the amount of time that a processor remains idle. However, larger values of *MAXDEPTH* also increase the stack management overhead. These two conflicting overheads need to be balanced. In the 15-puzzle, this balance occurs in the optimum range of *MINDEPTH* and *MAXDEPTH* values described earlier.

The range of optimal *MINDEPTH* and *MAXDEPTH* values should be similar for other simple search domains. If the node expansion cost is higher, larger values of *MAXDEPTH* can also be used without adverse effects, since the extra overhead of stack management becomes less significant. For shallow trees with high branching factors, it may be better to transfer multiple subtrees at a time, to ensure that the work transferred is significant compared to that retained by the sending processor. For more complex search domains, such as alpha-beta, information deficiency must also be considered.

5.4.5. Other Results

Kumar and Rao [12, page 13] found that, using their tree-splitting strategy in a shared-memory system (Sequent Balance 21000), there was little difference in performance for cutoff depths between 25% and 75% of the search depth (40 to 60). They also hypothesized that tree-splitting is superior to transferring single subtrees [12, page 8].

The experiments described in this thesis showed that, for *MINDEPTH*=0 (its best value), the best *MAXDEPTH* values ranged between 12 and 20, with 8 to 24 also giving reasonably good results. The optimal solution depths for problems 66 and 88 are 61 and 65, respectively; 8 to 24 translates to approximately 8% to 26% of the solution depth. The main differences between Kumar and Rao's program and PDFS were that:

- communication costs were lower for their program.
- they used slower processors.
- they split the tree; PDFS just sends out one subtree.
- their equivalent of *MAXDEPTH* is variable.
- they tested smaller problems.
- they implemented IDA*, which has extra overheads.

These differences must be considered when comparing their program to PDFS. The first two, and likely also the next two, differences make their results look better relative to PDFS; the last two make PDFS look better.

As can be seen by Figure 13, the optimal value of *MAXDEPTH* for problem 66 is approximately 25% of the solution depth. For problem 88, the optimal range is about 18% to 31%. The range is wider for problem 88 because the optimal solution

is deeper: small differences in *MAXDEPTH* are less significant for problems with deeper solutions. The curve is higher for problem 88 because the problem is bigger: the idle time at startup and termination is a smaller proportion of the total time than it is for the smaller problem. Also, the work loads rooted in the range from *MINDEPTH* to *MAXDEPTH* tend to be bigger, leading to proportionately fewer requests.

Finkel and Manber [7, page 245] also had good results for the 11-queens problem, another simple search domain.

5.5. Different-Sized Configurations and Problems

Tables 3 to 6 show the statistics for individual processors in 4, 8, 12, and 16-processor configurations, using Korf's four largest problems (#66, #60, #82 and #88) [10] as input. As before, the largest bound that avoids speedup anomaly was used, in order to avoid having the effects of speedup anomaly overshadow those of the overheads. Any set of *MINDEPTH* and *MAXDEPTH* values in the optimal range would give similar results; 0 and 16 were used here. Problems 60 and 82 are bigger than #66 and smaller than #88. Values that work well for #66 and #88 should work well for these problems also.

Each row in Tables 3 to 6 represents the data for one processor; the first row in each table represents the master processor. Since machines were selected only on the basis of availability (no other applications running on them), corresponding rows of different tables do not necessarily represent the same physical machine (i.e. the machine represented in row k for an n_1-processor configuration is usually not the same as that corresponding to row k in an n_2-processor configuration). Also, the set of machines used in a larger configuration does not necessarily contain all the machines used in a smaller configuration.

Table 3: Problem 66

Problem 66, 4 processors, 924074079 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
229988978	1323	1321	0.5	1.5	0.3	2	5	10	15
230011834	1322	1318	0.6	3.6	1.5	9	9	8	12
232373440	1322	1317	0.9	4.1	2.0	10	18	8	8
231699827	1323	1317	0.7	5.5	1.3	11	12	6	11
Problem 66, 8 processors, 924074079 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
117578361	674	668	0.8	5.2	2.5	16	13	17	35
114032033	671	655	1.1	15.1	2.2	12	16	18	31
115664283	672	663	1.0	8.4	2.7	15	34	18	27
114069190	672	660	1.0	10.5	2.8	18	22	15	31
115368674	672	661	0.9	9.4	2.0	13	48	15	28
114976903	672	660	1.0	10.7	2.7	15	75	11	28
115656337	672	658	1.1	12.7	2.7	16	27	12	34
116728298	672	658	0.8	13.0	2.2	12	10	11	36
Problem 66, 12 processors, 924074079 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
79262897	463	459	1.3	3.4	1.6	11	26	24	96
77265462	462	444	2.4	15.7	5.3	25	81	22	91
73232852	462	431	2.6	28.7	4.8	21	119	19	81
76580538	461	436	1.9	23.9	3.1	13	61	16	90
77974676	462	448	1.6	12.7	3.8	22	38	21	96
77440762	463	446	2.1	15.1	4.6	20	317	22	69
76942967	462	443	2.4	16.6	5.0	23	127	19	89
77441043	463	442	1.9	18.6	4.9	23	83	19	92
76802343	462	440	2.2	20.8	4.7	20	70	19	93
77410724	463	440	2.1	20.4	4.1	20	42	19	94
76825026	461	440	2.3	18.9	3.9	13	82	18	93
76894789	462	437	0.8	24.2	2.7	24	20	17	91
Problem 66, 16 processors, 924074079 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
61352185	362	350	3.5	8.6	6.1	25	105	28	94
59553372	361	343	2.5	15.2	4.7	20	68	26	92
54831234	361	325	2.4	33.2	3.8	14	83	19	97
56580250	357	326	2.7	28.9	5.1	18	109	18	95
55047180	361	310	13.8	36.4	4.5	22	83	15	98
55650858	360	318	2.8	39.6	5.5	23	80	15	98
55748694	360	318	3.0	39.7	5.4	20	130	17	92
55659127	360	318	3.1	39.2	5.0	18	114	15	93
60345892	361	347	2.7	11.0	4.7	20	160	26	86
59234603	361	343	2.1	16.0	5.5	28	76	24	93
59856536	361	342	1.6	16.7	3.7	24	35	24	96
58468162	361	337	2.5	20.9	4.4	18	71	24	94
58009657	361	332	2.4	26.0	4.3	22	68	20	98
57863677	361	332	2.8	26.3	5.3	25	95	19	95
57900530	360	332	2.2	26.3	4.1	16	44	22	96
57972122	359	332	2.5	25.0	4.7	19	176	20	89

Table 4: Problem 60

Problem 60, 4 processors, 1784841519 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
442883040	2573	2563	3.9	6.4	4.7	9	454	8	453
449306582	2572	2563	4.3	4.4	5.1	6	484	7	444
445188201	2572	2563	1.4	7.4	1.2	10	2	6	603
447463696	2572	2557	5.2	9.6	5.8	4	871	8	313
Problem 60, 8 processors, 1784841519 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
226230804	1306	1293	5.2	7.9	6.3	10	762	13	258
221699184	1305	1273	4.5	27.4	5.2	9	376	15	310
221305742	1306	1267	5.5	33.2	1.1	7	13	13	363
224045931	1305	1286	4.7	13.9	6.3	12	445	11	302
223479428	1305	1286	4.8	14.7	7.5	21	271	9	327
222299041	1305	1286	4.0	15.5	4.5	8	201	11	337
223380879	1305	1285	3.8	15.8	5.5	15	369	9	313
222400510	1306	1285	5.2	15.5	5.8	8	118	9	347
Problem 60, 12 processors, 1784841519 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
151875234	884	875	1.7	7.3	2.0	7	43	17	146
148601032	883	858	1.4	24.4	1.8	9	29	17	144
144980597	877	839	2.9	35.5	4.8	20	57	7	142
147704534	882	847	2.8	31.6	3.9	10	121	7	138
152206076	884	865	2.1	17.0	3.3	9	79	17	139
149806809	884	862	3.1	19.4	4.9	16	140	15	135
149500642	884	860	2.1	21.7	4.3	16	77	15	140
150184519	883	859	3.0	20.4	4.8	13	328	13	120
147845406	884	858	3.3	22.3	4.9	13	235	16	121
149381487	884	856	1.9	25.8	3.5	15	82	14	135
146867793	884	852	2.3	29.7	4.3	22	94	14	134
145887390	883	847	2.9	33.6	4.8	14	322	12	116
Problem 60, 16 processors, 1784841519 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
115740209	680	673	2.5	4.5	3.3	8	101	26	71
112811463	677	655	2.6	19.6	4.6	18	74	26	71
107179733	679	629	3.5	46.6	3.7	13	86	16	67
110781483	678	635	2.6	40.7	4.4	21	84	17	65
106777941	679	610	25.7	42.8	5.5	25	74	15	68
111238788	678	633	1.9	42.5	3.2	13	42	12	74
109049356	678	629	2.4	46.6	4.2	13	87	14	69
108286271	678	628	2.4	47.3	4.8	21	79	13	70
115469283	679	658	2.1	18.3	4.4	17	75	24	73
113303119	679	652	2.8	24.4	5.5	23	136	23	67
113371517	679	650	1.6	27.0	4.3	30	24	24	73
113618497	679	651	1.6	25.9	2.9	20	34	21	75
111762369	679	648	2.4	28.2	4.7	21	63	20	74
112516638	679	646	2.6	30.3	5.0	23	76	18	75
112315998	678	642	1.5	33.8	3.6	19	34	17	79
110618854	678	639	2.3	36.4	3.9	19	74	18	74

Table 5: Problem 82

Problem 82, 4 processors, 2790393007 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
697243011	3875	3864	4.2	7.3	5.0	10	477	12	393
697186103	3875	3863	4.7	7.2	5.7	10	453	10	401
698396047	3875	3864	1.4	8.9	1.1	9	3	10	550
697567846	3874	3858	4.6	12.4	5.5	10	722	7	313
Problem 82, 8 processors, 2790393007 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
349846012	1952	1944	3.5	4.2	4.7	11	326	17	223
344356770	1951	1915	3.6	32.6	4.9	8	427	17	203
350080394	1950	1940	3.1	6.4	4.4	15	238	15	231
349866477	1950	1935	3.9	10.5	5.3	13	309	12	223
349692527	1951	1936	2.4	11.7	3.1	9	159	12	245
350247713	1951	1936	2.3	12.3	3.7	14	217	8	240
347883455	1950	1933	3.0	14.6	4.7	14	183	10	243
348419659	1951	1934	1.1	15.6	2.5	15	13	8	269
Problem 82, 12 processors, 2790393007 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
234622347	1331	1310	17.4	4.2	21.0	12	1998	23	709
232578320	1330	1292	11.0	26.8	12.7	13	504	21	843
227016937	1330	1277	11.3	42.1	12.3	15	401	7	861
233722664	1330	1296	2.4	31.5	2.4	17	27	10	889
233293116	1330	1300	17.5	12.5	21.2	17	1127	16	791
232820234	1330	1297	17.9	15.1	21.9	16	1403	18	764
233635224	1330	1304	7.7	17.9	9.2	17	334	18	861
232258525	1330	1295	14.7	20.0	18.0	17	770	16	823
233291201	1330	1288	16.1	24.9	20.3	16	905	15	811
234830439	1329	1298	6.9	24.5	7.9	15	430	13	856
230039526	1329	1282	17.4	30.0	21.4	16	993	13	804
232284474	1330	1285	13.1	31.9	15.9	12	922	13	808
Problem 82, 16 processors, 2790393007 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
181179827	1012	998	5.9	7.8	7.7	12	351	38	232
175687686	1009	980	5.1	23.0	8.7	32	206	28	248
169673908	1010	959	4.6	46.5	6.5	32	103	25	247
174089878	1006	969	1.7	35.3	4.8	31	17	22	256
166253612	1009	918	41.5	48.7	9.6	25	402	22	231
172785932	1011	958	5.1	47.2	8.7	28	234	21	243
171943012	1010	959	5.8	46.1	8.3	19	243	20	243
173267844	1010	962	2.0	45.9	4.4	25	58	15	260
177854459	1011	983	6.2	22.2	9.0	32	346	30	236
177245471	1009	986	6.1	17.4	10.1	32	418	28	233
176150086	1010	985	3.7	21.2	6.4	24	174	28	247
175336264	1010	978	5.7	26.1	8.8	26	252	29	240
175859933	1010	978	6.2	26.4	8.8	17	351	31	232
174834688	1011	974	6.2	30.9	10.2	31	344	26	234
172871789	1010	972	5.2	33.4	8.5	31	178	27	244
175358618	1010	974	4.9	31.0	6.8	17	187	24	244

Table 6: Problem 88

Problem 88, 4 processors, 5156184395 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
1289096513	7380	7365	0.9	14.1	0.5	5	9	12	44
1282460225	7379	7370	1.0	8.3	1.6	14	16	10	38
1290031930	7379	7365	1.0	13.1	1.7	10	41	12	29
1294595727	7379	7357	1.3	20.1	2.2	13	65	8	23
Problem 88, 8 processors, 5156184395 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
646609767	3703	3697	1.5	5.0	2.5	12	28	21	57
632161532	3702	3644	1.7	56.3	3.3	21	175	18	33
648533361	3702	3694	1.6	6.3	3.1	18	52	19	50
645781060	3702	3687	1.5	13.1	3.7	21	35	19	52
652229477	3702	3691	1.3	9.6	3.4	20	13	17	56
646067820	3702	3689	1.2	11.3	3.1	18	45	18	51
643880079	3702	3689	1.4	11.5	3.3	21	35	15	53
640921299	3702	3690	1.0	11.0	2.1	13	19	17	55
Problem 88, 12 processors, 5156184395 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
435528005	2493	2482	3.4	7.9	4.8	14	74	18	159
426908277	2489	2446	1.7	41.0	2.6	15	50	20	153
421396853	2491	2442	4.5	44.6	3.2	6	236	15	138
431442757	2488	2460	3.3	25.3	5.3	17	128	14	148
436670810	2491	2474	3.3	14.0	4.9	15	172	20	141
432480342	2492	2468	3.2	20.6	4.4	13	163	14	148
431507382	2492	2466	2.7	22.8	5.4	25	96	14	153
431667275	2490	2468	2.3	20.3	4.2	20	53	16	155
422671042	2492	2442	3.3	45.8	4.4	13	103	12	155
430383472	2492	2465	3.6	23.4	4.9	10	107	13	153
426182680	2491	2461	2.9	27.0	4.4	18	191	16	140
429345500	2491	2458	3.4	30.1	5.6	19	388	13	125
Problem 88, 16 processors, 5156184395 nodes									
nodes	times (seconds)					requests sent		requests received	
	real	user	system	idle	wait	successes	failures	accepted	rejected
332498619	1889	1884	1.9	3.2	1.6	8	18	28	282
317182150	1888	1844	6.5	37.2	10.7	33	251	28	262
314610991	1888	1829	7.5	51.7	8.9	26	510	19	248
321118612	1887	1841	6.0	40.2	7.9	10	315	20	262
314519229	1888	1770	73.5	43.6	8.0	17	415	20	252
321720711	1888	1834	6.2	47.6	10.0	29	270	19	260
320860966	1887	1834	5.7	48.0	8.3	18	220	14	270
320190711	1888	1835	5.6	47.3	7.5	12	213	14	269
326843640	1888	1868	6.1	13.8	9.3	31	239	24	267
325599396	1887	1864	5.1	18.5	8.2	31	194	26	268
324350428	1888	1862	6.3	19.4	9.9	28	316	27	259
327569104	1887	1859	6.2	22.0	9.5	29	278	27	261
323383616	1888	1852	6.3	29.2	10.0	27	218	23	268
324626818	1887	1854	6.0	27.2	8.7	18	332	27	257
321821185	1888	1847	4.9	36.2	8.1	29	160	26	266
319288219	1888	1843	6.0	38.4	8.5	16	262	20	267

Real, user, system, idle and wait times mean the same as in Tables 1 and 2, except that they now refer to individual processors, instead of averages over all processors. The "requests sent" heading in Tables 3 to 6 is equivalent to the "work transfers" heading in Tables 1 and 2. The number of accepted requests is the number of times that a processor sent work in response to a request; the number of rejected requests is the number of times it did not (i.e. rejected the request). The total number of accepted requests is equal to the total number of successful ones because each work transfer requires a sender and a receiver. The total numbers of rejected and unsuccessful requests differ slightly because processors did not reply to messages that were received after, but sent before, termination was detected.

Table 7: Summary of times

Times (in seconds) for sequential and parallel programs						
Problem	sequential		parallel			
	fastest	slowest	4 processors	8 processors	12 processors	16 processors
66	5074	5325	1323	674	463	362
60	9873	10299	2573	1306	884	680
82	14913	15655	3875	1952	1331	1012
88	28386	29491	7380	3703	2493	1889

Table 8: Speedups

Speedups								
Problem	4 processors		8 processors		12 processors		16 processors	
	best	worst	best	worst	best	worst	best	worst
66	4.02	3.83	7.90	7.53	11.49	10.95	14.69	14.00
60	4.00	3.84	7.88	7.56	11.64	11.16	15.14	14.51
82	4.04	3.85	8.02	7.64	11.76	11.20	15.47	14.73
88	4.00	3.85	7.96	7.67	11.83	11.39	15.61	15.03

Table 7 contains a summary of the (real) times for the sequential and parallel programs for all four problems and Table 8 shows the corresponding speedups. Because of the differences in the timing measurements among machines, upper and

lower bounds for the sequential algorithm's time are included. The lower speedup bound uses the *user* time of the machine with the *shortest* sequential measurement; the upper speedup bound uses the *real* time of the machine with the *longest* sequential measurement. Corrupted measurements were not used for any of these timings. The upper bound can be as much as five percent more than the lower bound. In Table 8, the slower sequential measurements seem to suggest that super-linear speedup is possible; however, these high speedup figures are caused only by the differences in machine speeds. Super-linear speedup cannot occur in these experiments because the sequential and parallel algorithms search exactly the same nodes. The true speedup is somewhere between the best and worst shown in Table 8. A more accurate estimate of the true speedup can be obtained by running the sequential algorithm on all the machines, then using the average of these times as the sequential measurement.

As before, all machines were of the same type and speed and were initially idle; any sequential or parallel measurements that were corrupted by other processes that started later were redone. Since this set of experiments is intended to demonstrate the efficiency of PDFS using near-optimal values of *MINDEPTH* and *MAXDEPTH*, rather than to determine these values, stricter criteria were used to determine whether the results of experiments affected by non-PDFS processes should be discarded. The main basis for these criteria is to ensure that the effects of unrelated processes are small compared to the experimental errors in timing.

5.6. Observations

Tables 3 to 6 show that each processor spent negligible (less that 2% in Tables 3 to 6) time looking for work, which shows that further improvements caused by changing *MINDEPTH* or *MAXDEPTH* to reduce message frequency will be slight. Other possible improvements are discussed in Chapter 6. Processors were busy most

of the time because the search tree is large: after receiving a subtree, a processor can continue searching for a long time without having to look for more work. For most of the search, there is plenty of work for idle processors to find. If the search space is so small that the idle times at startup and termination are significant, then the degree of parallelism is too high and the benefit of adding more processors is questionable.

Another observation (see Tables 3 to 6) is that, in each run, each processor searched approximately the same number of nodes. (Processor 0 often searched a few more nodes because it started earlier than the slaves, which it had to create.) On the other hand, when *MINDEPTH* and *MAXDEPTH* were poorly chosen, especially when they were close together, one processor, usually processor 0, issued fewer requests and searched many more nodes than the others (figures not shown). This difference indicates that one processor, usually processor 0 because it initially has the entire tree, has a large work load that cannot be subdivided because *MAXDEPTH* is too small.

For large *MINDEPTH* values, the difference in node counts indicates that processor 0 retains most of the tree and only sends small parts of it to other processors, which soon become idle again. There are two solutions to this problem. One solution is for idle processors to try to get work from processor 0 first whenever becoming idle. This solution should work well for centralized algorithms, but the master would become a bottleneck in large configurations. The other solution, which is more appropriate for decentralized algorithms is to send out larger subtrees (reducing *MINDEPTH*) in order to distribute work more evenly, thus almost eliminating any differences between the master and the slaves.

When appropriate *MINDEPTH* and *MAXDEPTH* values were chosen, this effect sometimes occurred in slave processors also, but usually to a lesser extent. This observation suggests that one machine did not receive many requests while

searching the shallow depths of a large subtree. This machine searched more nodes than any others, supporting the claim that it searched a large subtree relatively unimpeded while the others struggled to find work.

In the 16-processor configurations, the system time for the machine corresponding to the fifth row is much higher than for other machines. This machine always displays higher system time figures, even for the sequential program; PDFS does not treat this machine differently. The effect of this extra system time is small, however: the variations in speeds among individual machines often caused larger differences in the node count figures.

5.7. Conclusions

Tables 7 and 8 show that good speedups can be obtained, despite the simple work distribution strategy, the low node expansion cost, and the high communication costs. However, near-linear speedups should not be considered unusual for a simple problem domain like the one used in this thesis. The same set of nodes is expanded regardless of the order in which successors are searched.

The choice of values to use for *MINDEPTH* and *MAXDEPTH* depends on the type of problem. Different values might be more appropriate for different tree structures (e.g. different branching factor, different depth, no narrowing at deep levels). However, based on the two examples of the 15-puzzle, which differ in size by a factor of more than 5.5, setting *MINDEPTH* to 0 and *MAXDEPTH* to 25% of the maximum search depth (the maximum search depth is usually more easily predetermined than the solution depth) seems to give good results, at least for exhaustive problems with low node expansion cost. Setting *MINDEPTH*=0 and *MAXDEPTH*=16 (reasonably close to 25% of the search depth) gave good performance for both problems, as well as two that are intermediate in size.

More complex work distribution strategies are necessary in domains in which information deficiency can be a problem (e.g. branch-and-bound, including alpha-beta). The extra search effort resulting from information deficiency must be balanced against the other overheads, and thus the speedups should be lower.

In problems with higher node expansion costs, stack management and flag checking become insignificant compared to the other operations involved in node expansion (i.e. the rise in Figures 11 and 12 as *MAXDEPTH* increases should be smaller). If the overheads remain constant, speedups should increase because more useful work is done. Thus, *MAXDEPTH* can be increased if the node expansion cost is higher, since the problem is affected less by stack management, but can benefit from the extra freedom in assigning work.

However, the suggested formulas for determining values of *MINDEPTH* and *MAXDEPTH* for the example problem domain appear to be good for two problems of the same type, but of substantially different size. Perhaps such a general formula can be found for other problem domains also, although the formulas for determining *MINDEPTH* and *MAXDEPTH* may be different. For a complex problem domain, it might be worthwhile to start by trying the above methods for guessing *MINDEPTH* and *MAXDEPTH*, then testing the program on small problems and making any adjustments that improve performance, then using the adjusted formulas for larger problems.

CHAPTER 6

6. Conclusions and Future Additions

6.1. Larger Configurations and Connectivity

In some distributed systems, the hardware does not provide direct communication between all pairs of processors; messages can be sent only to neighboring processors. Such a restriction is necessary in large systems because it is not feasible to provide N^2 connections for N processors. There are two options in such systems. One option is to simulate complete connectivity by having intermediate processors forward messages to their destinations; sending these extra messages increases the time required for communication. The other option is to send messages only to neighboring processors; this restriction increases starvation if work is available on distant processors, but not on adjacent ones. However, the reduced connectivity may also decrease the number of interruptions received by busy processors, as well as the total amount of network traffic.

6.2. Better Allocation of Work

Currently, a unit of work consists of a single subtree, whose size is controlled only by the two constants, *MINDEPTH* and *MAXDEPTH*. Instead of remaining constant, these values could depend on the expected size, as predicted by a heuristic function, of the initial problem or transferred subtree. The experiments have shown that larger subtrees are preferable. Setting an upper bound on the heuristic evaluation of nodes that can be transferred might be a better way to force larger subtrees to be transferred, since subtrees rooted at the same depth may vary substantially in size.

Even without this limitation, the heuristic can be used to predict which of the available subtrees is expected to be bigger. (Using a heuristic to distinguish between siblings is equivalent to move ordering, which can reduce unnecessary search in many applications, such as two-player games [24].)

Sending multiple subtrees in response to a single request for work might result in more even work distribution, thus resulting in fewer requests. Such a change, including an approximately equal split (by sending out half of the nodes at each depth) as Kumar and Rao did [12], would be easy to make in PDFS.

Starting with a reasonable static partition of the search tree will eliminate much of the startup overhead, since all processors would be busy initially. If the initial static partition is good, all processors should be able to search large subtrees without being interrupted by requests until much of the search is finished. However, since it is almost impossible to partition the tree equally, processors will eventually become idle and start looking for work.

Another possible improvement would be to have idle processors request work from several processors at once without waiting for a reply after each request. This change would reduce the amount of time required for an idle processor to find work, but if most processors have work to give (as is usually the case), more busy processors will be interrupted unnecessarily. If several processors send work, extra subtrees can be queued and later searched or reassigned; however, this method increases the number of parallel tasks, which unnecessarily increases other overheads (especially information deficiency in alpha-beta).

6.3. Reducing the Node Expansion Cost

Checking the two flags, which indicate whether any messages are waiting and

whether another processor has solved the problem, slightly increases the cost of a node expansion. If desired, these checks can be eliminated by taking the necessary action (calling the polling routine or terminating the process) within the interrupt (signal) handler. If these two flags are removed, expansion of nodes deeper than *MAXDEPTH* takes no more time for PDFS than for the sequential algorithm. The flag indicating that another processor found a solution was, in fact, removed from the subroutine searching deeper nodes, since solutions were avoided to eliminate the possibility of speedup anomaly. If a solution were to be found without either this flag or the modification to the interrupt handler, busy processors would continue searching their subtrees until the search backtracks to a node shallower than *MAXDEPTH*.

The stack management overhead can be offset by using global variables instead of parameters, thus reducing the amount of implicit stacking in the recursive procedures. Alternatively, the recursive algorithm could be converted to an iterative one, which requires explicit stacking anyway, but may be faster. The only extra stack overhead in the parallel program would then be the extra information (e.g. set of successors that have not yet been transferred) that needs to be stacked. In addition, it may also be possible to speed up some of the operations used for evaluating a node or generating its successors.

Some of these changes may also improve the sequential algorithm; in this case, they should be made to both algorithms. The speedups should decrease because the total time decreases while the overheads remain constant. Let S be the total time spent searching nodes, which is also equal to the sequential search time (assuming that both algorithms search the same nodes). Let P be the total time spent on parallel overheads. Suppose the node expansion cost is reduced by a factor of α ($\alpha>1$) in both the sequential and parallel algorithms. The old speedup, for N processors, was $\frac{S}{S/N+P} = \frac{NS}{S+NP}$; the new speedup is $\frac{S/\alpha}{S/(\alpha N)+P} = \frac{NS}{S+\alpha NP}$, which is less.

6.4. Transpositions

In some problem domains, identical states are often reached via two or more different paths. When a state identical to a previously searched state is found, a *transposition* is said to occur. Detecting transpositions, especially those near the root, can substantially reduce the amount of search required.

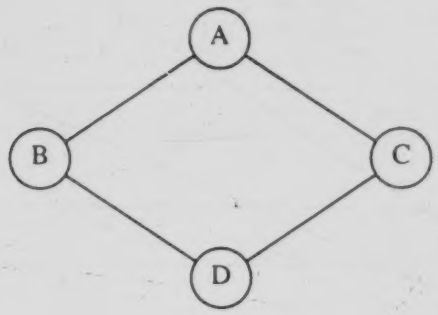


Figure 14: A transposition

In Figure 14, node D is generated twice: once as a successor of B and once as a successor of C.. If this transposition is not detected, the subtree (which may be large) rooted at D will be searched twice, unless a solution is found.

Transpositions are detected by storing states (not necessarily all of them; some entries may be overwritten if the table is too small), as they are encountered, in a large table. This table is checked before searching successors of a state. If a state is already in the table, and information associated with the state is sufficient (i.e. re-searching will not provide more information), the search is cut off, since an identical subtree has already been searched. Since most problems are much too big to allow transposition tables to contain all nodes encountered, there should be a strategy to decide which nodes are likely to lead to transpositions or larger cutoffs.

Detecting transpositions increases the cost of node expansion, but can also substantially decrease the number of nodes searched, especially for transpositions occurring near the root. However, in some problem domains, such as the 15-puzzle, transpositions occur infrequently and the cost of searching a node is small. The overhead of managing the transposition table exceeds the savings provided by the reduced number of nodes [22].

In a parallel environment, transposition tables cause additional overheads, and therefore lower speedups should be expected. In a shared memory system, the tables can create a bottleneck if many processors try to access them simultaneously, since processors must be delayed because of hardware limitations and mutual exclusion requirements. In a distributed system, each processor usually has its own local table, which only contains transpositions found by that processor, but may be updated periodically to include other transpositions. Since one processor cannot access another processor's table, some transpositions detected by the sequential algorithm might be missed by the parallel algorithm. This re-searching of subtrees, a form of information deficiency overhead, can be reduced by periodically sending lists of recently added nodes; however, these extra messages add to the communication overhead. This situation results in a tradeoff between information deficiency and communication overheads; the appropriate compromise depends on the frequency of transpositions, which depends on the problem domain.

Another option is to simulate shared memory by having one machine function as a table manager [23, page 102]. Entries are added and checked by sending messages to this machine. There are three problems with this method: the communication overhead is very high, processors are idle while they wait for responses, and a table manager can become a bottleneck in large systems (with hundreds of processors). If a query to the table manager is made for every node, the high idle time becomes a

severe problem, even if a processor checks its local tables while waiting for a reply from the table manager. Schaeffer [23, page 102] recommends using a table manager only for transpositions occurring near the root because detecting these transpositions saves the most work.

A processor that becomes aware of a transposition found by another processor can add the transposition to its local table, thus reducing the need for a table manager. The few transpositions occurring near the root can be broadcast to all processors.

The bottleneck can be reduced by using multiple machines as table managers. A hashing function could be used to determine which machine should be used to store the transposition. However, each table manager uses resources that could otherwise be used to perform additional search.

It should also be noted that, in many search algorithms, transposition tables are maintained implicitly. For example, A*'s *CLOSED* list includes the functions of a transposition table because nodes are not re-expanded if they appear there. In problems in which moves are reversible (e.g. 15-puzzle), IDA* does not consider the reversal of the last move (i.e. the transition leading back to the parent node). The elimination of such moves, which saves much unnecessary search, is a limited form of transposition detection.

6.5. Other Search Algorithms

PDFS, although presently implemented for the simple problem domain described in Section 5.1, can be extended to do more complex searches. In the example used in this thesis, each processor has complete information: the initial state, the threshold, all moves in the search path, and all unexpanded successors of nodes in the search path. Processors need to communicate only to distribute work and to signal

completion; they do not need to return results to other processors. (A processor that solves the problem can immediately output the complete path to the solution, then tell the other processors to stop.)

This section describes the modifications needed to use PDFS for IDA*, simple branch-and-bound, and alpha-beta. Because of extra synchronization, messages, and information deficiency, PDFS's speedups should be lower when implementing these algorithms, especially alpha-beta.

6.5.1. IDA*

IDA* can be implemented by repeating the example algorithm for each iteration. In general, it is also necessary to determine the smallest heuristic value of the nodes that cause cutoffs, in order to determine the threshold for the next iteration, although in some problems, the next threshold is easy to determine.† Each processor can keep track of the lowest-cost cutoff that it encounters. At the end of the iteration, the master processor can calculate the minimum of these values to determine the next threshold.

The performance for IDA* should still be good, as indicated by Kumar and Rao's results [12]. The only extra overheads are that the idle times at startup and termination occur in each iteration, and that termination detection must be repeated for each iteration; however, these overheads are small.

Another approach is *parallel window search* [20], in which several iterations proceed simultaneously. When one processor finds a solution, all processors using larger thresholds can stop. Processors using smaller thresholds must continue because a cheaper solution may exist. However, for many problems, determining

† In the 15-puzzle, the next threshold is always 2 more than the current one.

suitable thresholds in advance is difficult.

6.5.2. Branch-and-Bound

In depth-first branch-and-bound, which can be used for the traveling salesman problem [13] (although better algorithms exist) the bound changes frequently. One processor may find a good bound while others are still using poor ones, thus resulting in unnecessary search. One solution would be to transmit new bounds. The frequency at which bounds should be transmitted depends on the cost of transmission and on the number of cutoffs that are caused by the new bound: more frequent transmission reduces information deficiency, but increases communication overhead.

Even though it changes, the bound is still global, like IDA*'s threshold: any processor's bound can be used by any other processor. A processor that finds a good bound can immediately broadcast it to all other processors. A processor can use such a new bound immediately if it is better than the current bound. Shared memory, even if limited, would be good for storing the bound.

As in IDA*, each processor can proceed independently, without having to return results to other processors. The only differences are that bounds should be transmitted (transmission of bounds does not depend on which nodes are being searched; it can occur at any time) and that, at the end, an optimal solution has to be chosen from among those found by different processors. PDFS can be modified to do depth-first branch-and-bound simply by sending new bounds to all processors at certain times (for example, when they improve the current best global bound by a certain amount).

6.5.3. Alpha-Beta

In alpha-beta, one processor's alpha and beta bounds may not be applicable to

other processors. When the bounds are applicable, they are often inferior to those available to a sequential program. This problem occurs because older siblings, which are searched before younger ones in a sequential program, but at the same time in a parallel program, can no longer provide better bounds for younger siblings. Processors are unable to take full advantage of the savings provided by the alpha-beta algorithm over plain minimax. This information deficiency can be reduced by constraining the choices of subtrees to send, at the cost of increased idle time.

In what is now called the *mandatory work first* approach [1], only work that the sequential algorithm must do anyway (the *minimal tree* [9]) is parallelized initially. (There is no information deficiency in this initial stage.) After the minimal tree has been searched, other nodes are searched in parallel, with some extra search being caused by information deficiency.

Since extra search is preferable to simply keeping processors idle, and each extra processor contributes less to the (useful) search effort, at some point it may be better to use extra processors for *speculative computing* [23, page 109]: computations that may turn out to be unnecessary, but might instead save much work.

In principal variation splitting (PVSplit) [14][15] (a synchronous, centralized algorithm that performs well on strongly-ordered [14] game trees), a subtree is reassigned only if an older sibling has already been searched (Feldman *et al.* [6] call this technique *younger brothers wait*). Move ordering increases performance greatly because it reduces extra search caused by information deficiency. Iterative-deepening [24] can be used to improve move ordering. However, PVSplit suffers from high synchronization overhead. Schaeffer [23, page 96] proposed *dynamic* principal variation splitting, in which *controller* processes can tell idle processors to help busy ones. Dynamic PVSplit overcomes much of the synchronization overhead of PVSplit, at the expense of increased information deficiency and communication overheads.

Schaeffer used dynamic PVSplit in his chess program, *Phoenix*. Dynamic PVSplit is similar to PDFS, except that PDFS does not use controller processes and PDFS currently cannot handle alpha-beta efficiently. Vornberger and Monien [26] describe additional rules that PDFS can use to reduce information deficiency:

- Successors are sent in the same order as they are generated by the move ordering routine.
- The first successor is never sent; it is always searched by the processor that is searching its parent.
- A successor is never sent unless one of its siblings has already been evaluated (''younger brothers wait'').

Also, unlike in IDA*, a processor needs to return information, such as the minimax value of the subtree, to the processor from which it received work; this information can cause more cutoffs in descendants of unsearched sibling nodes. This requirement forces a master processor (that has finished searching successors that have not been transferred) to wait, perhaps for a long time, until all of its slaves have finished, before returning to the parent of the current node. It is also beneficial to send slaves new bounds as they become available and to terminate slaves if cutoffs occur.

While it is waiting for other processors to finish, a processor can obtain work from elsewhere. Feldman *et al.* [6] recommend (the *helpful master* concept) that the processor helps one of its slaves instead of taking work from elsewhere. One advantage of the helpful master concept is that a slave, for which the master is waiting, will finish sooner, thus reducing the time that the master has to wait. Another advantage is that the new work load is always rooted deeper in the tree, thus putting a limit on the number of unfinished jobs that may be queued as processors wait for their slaves (since the slave that the master is helping may itself become idle and try to help its

master).

Also, as slaves finish, they can provide better bounds for their siblings, or even discover that their siblings no longer need to be searched. The master can forward this information, which it receives when slaves terminate, to its remaining slaves, which can then update their bounds or terminate, whichever is appropriate.

Generally, for alpha-beta, sending smaller subtrees should decrease the amount of extra search caused by information deficiency. When looking for work to send, a processor should check deeper nodes first (i.e. send out the deepest-rooted instead of shallowest-rooted subtree). Increasing *MINDEPTH* may also help by ensuring that large, shallow-rooted subtrees are searched one at a time.

6.5.4. Other Algorithms

The work distribution method described in this thesis can be used for many problems that can be expressed recursively with more than one recursive call per function. The main exception is when the results of one recursion are required before the program can proceed to the next one.

Other types of search can also be parallelized. Staged searches that start by using A*, then switch to IDA* when memory is full can be parallelized (at least in the second stage) by dividing the *OPEN* list up among the processors. The additional memory provided by extra machines can also be used for larger *OPEN* lists or bigger transposition tables.

6.6. Generalization

PDFS was designed to be general: much of the code can be reused when implementing a different search algorithm. However, the speedups depend greatly on the

search domain, especially if information deficiency is possible. With minor modifications, PDFS can be used as a framework for parallel depth-first search in general, as opposed to one single algorithm. (Some modularity was sacrificed to improve efficiency.) Users would have to write their own domain-dependent search routines, which would be similar to the sequential program, but would have to call two built-in subroutines: one that checks if successors have been reassigned and one that checks for and replies to messages. The user would not have to worry about process creation, message passing, deadlock, termination detection, etc. For branch-and-bound searches, such as alpha-beta, the user may try to improve performance by providing a work splitting algorithm, which would try to assign nodes in a way that reduces information deficiency.

Finkel and Manber's DIB program [7] automatically handles process creation and work distribution. The user need only provide problem- and domain-dependent information, such as the initial state and the operations required for node expansion (for example, node evaluation and successor generation).

The performance of such general purpose programs is unlikely to be as good as for specialized programs, which can take advantage of domain-specific features. Also, general-purpose programs require much modularity, which must often be sacrificed to improve performance. This fact is especially true in applications like alpha-beta, in which efficiency depends greatly on the work selection strategy and the communication of additional information (bounds).

6.7. Fault Tolerance

In the current version of PDFS, if one processor crashes, any processor that requests work from it will also crash. Eventually, all processors will crash and the entire problem must be redone. This risk is higher in larger systems, since there are

more processors that can crash. In order for a program to recover, it is necessary to know whether a subtree assigned to another processor was actually searched, or just lost because of a crash. Therefore, it is necessary to maintain redundant copies of important information so that it is not lost if one copy is destroyed. Information is considered to be important if it can avoid the need to redo large amounts of work. It is advantageous to restart crashed processors quickly to reduce the amount of idle time. Also, all processors must be informed of the existence of restarted or additional processors.

Such information can be maintained by requiring a processor that finishes searching a subtree to send the results of that search back to the processor from which the subtree was received. Another option is to have one central processor, but this processor could become a bottleneck and there are additional problems if that processor crashes. In DIB [7, page 247], fault tolerance is implemented by having processors perform redundant searches when they cannot find anything else to do.

6.8. Loaded and Heterogeneous Systems

For practical purposes, as opposed to experimentation with timing, it is not necessary that all machines be idle or run at the same speed. A slower or heavily-loaded machine, which allocates less CPU time to PDFS, cannot search as many nodes in same amount of time as a fast, idle machine, and thus transfers more nodes to machines that become idle sooner.

For example, if there are two identical machines, A and B, with A idle (except for a PDFS process) and B loaded with one other CPU-intensive process with the same priority, PDFS will get all of A's CPU time and half of B's; A will search about twice as many nodes as B (approximately two-thirds of the search space if there are no other machines). The results are similar if B is idle, but only half as fast as A.

In a heterogeneous system, there may be differences in the internal representations of data types. These differences, if present, require that data in messages be converted to a standard representation. Although this conversion increases the communication cost, the extra overhead is small because only a small proportion of the total time is spent on communication anyway.

By reducing the priority of the processes, PDFS can effectively utilize the idle time in all machines in a large network without significantly delaying other applications. This load balancing is inherent in the asynchronous work distribution scheme; no extra code is necessary. Also, since low-priority processes adapt to changing load conditions, it is not necessary to move them off busy machines: each machine can simply run one PDFS process, which automatically gets less CPU time as the machine's load increases.

6.9. Conclusions

This research has demonstrated that an efficient, fully-distributed implementation of depth-first search is possible. The experiments have shown that PDFS's efficiency depends greatly on the size of a unit of work. Varying *MINDEPTH* and *MAXDEPTH* is a simple, general, though coarse, way of controlling the sizes of subtrees that are transferred. *MINDEPTH* and *MAXDEPTH* adjustments can be used to improve PDFS for specific problem domains.

It is unfortunate that a much larger system, with more than 100 identical processors, was not available. In small configurations, the advantages of decentralized algorithms like PDFS are often outweighed by their additional overheads because bottlenecks are less likely to occur. To investigate the effects of different tree structures (e.g. branching factor and depth), PDFS could be tested on other simple problems (e.g. N-queens), instead of more examples of the 15-puzzle; little modification

would be required for problems that do not suffer from information deficiency. However, because much computer time is needed and all machines must be idle (for accurate timing measurements), there was not enough time to test PDFS on other problem domains. Even without using more problem domains, most experiments had to be delayed or repeated several times because other applications were using the machines.

Since precise timing is usually not required in real applications, extended versions of PDFS could be set up to allow large artificial intelligence applications to run on machines that are sometimes unused. PDFS and similar programs, running at low priority, can effectively utilize the idle time in large networks of computers, while minimizing the additional load on busy machines.

References

1. S. G. Akl, D. T. Barnard and R. J. Doran, Design, Analysis and Implementation of a Parallel Tree Search Algorithm, *IEEE Transactions on Pattern Analysis and Machine Intelligence 4*, (1982), 192-203.

2. T. Breitkreutz, S. Sutphen and T. A. Marsland, Developing NMP Applications, Tech. Rep. 89-11, University of Alberta Department of Computing Science, March 1989.

3. M. S. Campbell and T. A. Marsland, A Comparison of Minimax Tree Search Algorithms, *Artificial Intelligence 20*, (1983), 347-367.

4. E. W. Dijkstra, W. H. J. Feijen and A. J. M. Gasteren, Derivation of a Termination Detection Algorithm for Distributed Computations, *Information Processing Letters 16*, (1983), 217-219.

5. V. Faber, O. M. Lubeck and A. B. White, Jr., Superlinear Speedup of an Efficient Sequential Algorithm is not Possible, *Parallel Computing 3*, (1986), 259-260.

6. R. Feldman, B. Monien, P. Mysliwietz and O. Vornberger, Distributed Game Tree Search, in *Parallel Algorithms for Machine Intelligence and Vision*, V. Kumar, P. S. Gopalakrishnan and L. Kanal (ed.), Springer-Verlag, 1990, 66-101.

7. R. Finkel and U. Manber, DIB—A Distributed Implementation of Backtracking, *ACM Transactions on Programming Languages and Systems 9*, (1987), 235-256.

8. M. J. Flynn, Very High-Speed Computing Systems, *Proceedings of the IEEE 54*, (1966), 1901-1909.

9. D. E. Knuth and R. W. Moore, An Analysis of Alpha-Beta Pruning, *Artificial Intelligence 6*, (1975), 293-326.

10. R. E. Korf, Depth-First Iterative-Deepening: An Optimal Admissible Tree Search, *Artificial Intelligence 27*, (1985), 97-109.

11. R. E. Korf, Macro-Operators: A Weak Method for Learning, *Artificial Intelligence 26*, (1985), 35-79.

12. V. Kumar and V. N. Rao, Scalable Parallel Formulations of Depth-First Search, in *Parallel Algorithms for Machine Intelligence and Vision*, V. Kumar, P. S. Gopalakrishnan and L. Kanal (ed.), Springer-Verlag, 1990, 1-41.

13. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys, eds., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, 1985.

14. T. A. Marsland and M. Campbell, Parallel Search of Strongly Ordered Game Trees, *Computing Surveys 14*, (1982), 533-551.

15. T. A. Marsland and F. Popowich, Parallel Game-Tree Search, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, (1985), 442-452.

16. T. A. Marsland, T. Breitkreutz and S. Sutphen, A Network Multi-Processor for Experiments in Parallelism, *Concurrency: Practice and Experience 3*, (1991), 203-219.

17. N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

18. D. Parkinson, Parallel Efficiency can be Greater than Unity, *Parallel Computing 3*, (1986), 261-262.

19. J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem*

Solving, Addison-Wesley, 1984.

20. C. Powley and R. E. Korf, Single-Agent Parallel Window Search, *IEEE Transactions on Pattern Analysis and Machine Intelligence 13*, (1991), 466-477.

21. V. N. Rao, V. Kumar and K. Ramesh, A Parallel Implementation of Iterative-Deepening A*, *Proc. Sixth National Conference on Artificial Intelligence*, 1987, 178-182.

22. A. Reinefeld and T. A. Marsland, Memory Functions in Iterative-Deepening Search, Mitteilung (Technical Report) FBI-HH-M198/91, Universität Hamburg Fachbereich Informatik (University of Hamburg Department of Computer Science), May 1991.

23. J. Schaeffer, Distributed Game-Tree Searching, *Journal of Parallel and Distributed Computing 6*, (1989), 90-114.

24. D. J. Slate and L. R. Atkin, CHESS 4.5—The Northwestern University Chess Program, in *Chess Skill in Man and Machine*, P. W. Frey (ed.), Springer-Verlag, 1977, 82-118.

25. G. C. Stockman, A Minimax Algorithm Better than Alpha-Beta?, *Artificial Intelligence 12*, (1979), 179-196.

26. O. Vornberger and B. Monien, Parallel Alpha-Beta versus Parallel SSS*, *Proc. IFIP Conf. on Distributed Processing 3*, (1987), 613-625.